IMPARARE PYTHON

di Roberto Allegra

Contenitori

Controllo del flusso

Funzioni e moduli

Classi ed eccezioni

INTRODUZIONE

E così avete deciso di imparare a programmare in Python. Ottima scelta! Se non avete mai programmato prima, Python è probabilmente il linguaggio migliore per iniziare: è semplice, chiaro, potente e versatile. E, soprattutto, è divertente: grazie all'interprete potrete chiarire i vostri dubbi scrivendo codice interattivamente, ottenendo una risposta immediata; la funzione help potrà chiarirvi subito il significato di parole chiave, moduli e classi; gli oggetti builtin e la corposa libreria standard, infine, vi permetteranno di ottenere il massimo risultato con il minimo sforzo. Questo libro vi introdurrà a queste ed altre operazioni, e alla fine della lettura sarete pronti a scrivere script e programmi, in modo OOP e pulito. Se, invece, venite da linguaggi di più basso livello, come C e C++, Python vi stupirà con effetti speciali che difficilmente avreste creduto possibili, come ad esempio la capacità di invocare dinamicamente classi e funzioni a partire da semplici stringhe, di creare tipi a run-time e di analizzarne la struttura attraverso dei comodi dizionari. Se avete sempre programmato con linguaggi a tipizzazione statica (come C++, Java e C#), preparatevi a un mondo in cui le variabili non vanno dichiarate prima dell'uso, in cui non è necessario conoscerne a priori il tipo, quanto piuttosto verificare che implementino il protocollo che volete usare. Questo libro è pensato anche per voi: di quando in quando, segnalerò alcune somiglianze e discrepanze fra Python e altri linguaggi, e soprattutto gli errori più comuni in cui tendono a incorrere i programmatori provenienti da altri mondi, quando tentano di "tradurre" istintivamente i propri schemi mentali in Python. Nel corso della lettura acquisirete una panoramica chiara dei fondamenti del linguaggio: dall'uso dell'interprete come calcolatrice, fino alla programmazione orientata agli oggetti, passando per moduli, funzioni e tipi fondamentali. Questo è quello che troverete in questo libro, ma è altrettanto utile chiarire quel che qui dentro non troverete. Questo testo non è una reference completa su Python e, soprattutto, sulla

sua sterminata libreria standard. Sarebbe impossibile riuscirci in sole 160 pagine, o ancor peggio, ci si ridurrebbe ad uno scarno e inutile dizionarietto sintattico. Ho quindi evitato di trattare funzionalità del linguaggio troppo avanzate, per esperti, o meno comuni, per concentrarmi su un approccio fortemente pratico, fatto di codice, descrizioni e suggerimenti. Alla fine della lettura (o parallelamente!) potrete dedicarvi a testi più impegnativi, come quelli elencati nella sezione Bibliografia. Non troverete neanche molte note storiche sul linguaggio, sul "quando" una data funzionalità è stata proposta e inserita: per un libro con queste finalità sarebbero informazioni utili solo a confondere le idee. Tutte le caratteristiche trattate, invece, fanno parte della versione di Python più recente al momento della scrittura di questo libro: la 2.5.
Nella stesura degli argomenti ho cercato di essere quanto più scrupoloso e attento possibile, ma questo non è il mio primo libro, e ormai so per esperienza che qualche errore finisce sempre in sede di stampa. Pertanto, la prima cosa che vi consiglierei di fare prima di affrontare la lettura è di andare nel sito www.robertoallegra.it. Troverete lì eventuali errata corrige, segnalazioni e approfondimenti, nonché i codici sorgenti degli esempi proposti in questo libro. Non mi resta che augurarvi buona lettura, buono studio e soprattutto… buon divertimento!

I PRIMI PASSI

1.1 QUALCHE INFORMAZIONE SU PYTHON…

Cominciamo a definire cos'è Python: si tratta di un linguaggio creato nei primi anni novanta da un programmatore olandese, **Guido van Rossum**. Per le sue doti di espressività, flessibilità e chiarezza, Python ha attirato un'ottima comunità di sviluppatori e ha subito nel corso degli anni numerosi miglioramenti, fino ad arrivare a vantare fra i suoi utenti società del calibro di Google e Youtube, ed enti come il MIT e la NASA. Da un punto di vista strettamente tecnico, Python è un linguaggio **open source, cross-platform, multiparadigma, interpretato e a tipizzazione dinamica e forte**. Ha una ricca **libreria standard** e si presta in modo naturale ad essere utilizzato per lo sviluppo di **script e prototipi**, nonché come **linguaggio collante**. Se avete capito quest'ultimo paragrafo nelle sue implicazioni più profonde, probabilmente siete già dei programmatori esperti; se non ci avete capito niente, non preoccupatevi: è normale. Qui di seguito trovate una serie di domande e risposte fatte apposta per chiarire i termini in neretto (leggetele anche se siete esperti: ne approfitterò per cominciare a introdurre elementi essenziali del linguaggio).

1.1.1 'Open Source' significa gratuito?

Significa gratuito e tante altre cose, fra cui libero e non-proprietario. Potete scaricare i sorgenti e gli eseguibili di Python liberamente. Potete scrivere i vostri programmi in Python e rilasciarli senza dover pagare nulla ad alcuna società. Siete liberi di scrivere un vostro interprete e metterlo in commercio, e per farlo potete addirittura basarvi sul codice stesso di Python, modificarlo e rivenderlo, senza alcun obbligo di rilasciarne il codice sorgente. In un mondo in cui ogni giorno viene sfornato l'ennesimo linguaggio proprietario, queste libertà sono preziose. Se siete interessati ad approfondire ulteriormente il discorso, potete leggere la Python Software Foundation License, che può essere considerata una variante (pienamente compatibile) della licenza GPL.

1.1.2 Cosa vuol dire Cross-Platform?

Significa che i programmi scritti in Python gireranno senza bisogno di alcuna modifica su molti Sistemi Operativi. Questi includono le piattaforme più utilizzate: Windows, Linux, e Mac OS X, ed è da notare che in quest'ultimo (così come in molte versioni di Linux) Python viene già fornito nell'installazione di base. Ma oltre a questi sono stati realizzati porting di Python per le piattaforme più disparate: da Palm OS e Windows CE, fino a Solaris e QNX, passando addirittura per Amiga, Nintendo DS e Playstation! È chiaro che la compatibilità è assicurata fin quando userete la parte più "standard" del linguaggio, e non darete per scontate componenti che invece variano a seconda del Sistema Operativo utilizzato, come ad esempio la struttura di file e directory. Inoltre, per permettere funzionalità specifiche, Python mette a disposizione anche dei moduli che si basano direttamente su un singolo SO: in questi casi, la documentazione chiarisce sempre se sia possibile utilizzare i moduli anche nelle altre piattaforme che avete scelto come vostra destinazione, eventuali effetti collaterali nel passaggio fra un SO e l'altro, e possibili alternative cross-platform. Ma a parte di questi casi limite, di solito non dovrete fare proprio nulla per assicurare la compatibilità della vostra applicazione: rilasciatene i sorgenti e chiunque potrà eseguirli.

1.1.3 Cosa significa 'interpretato'?

In informatica, esistono due sistemi per l'esecuzione dei programmi: la **compilazione** e l'**interpretazione**. Nei linguaggi nati per essere compilati (come C, C++ e Pascal) i programmi vengono tradotti in linguaggio macchina da un software apposito, detto **compilatore**. Il risultato di questo processo è un file eseguibile (ad esempio, un .exe, sotto Windows), che viene poi distribuito senza bisogno di accludere i sorgenti. I programmi scritti in un linguaggio nato per essere interpretato (come Python, o il BASIC), invece, non subiscono questo processo di trasformazione "una tantum": ogni volta che il programma viene eseguito, i sorgenti vengono

letti riga per riga da un programma detto **interprete**, che provvede ad eseguirli al volo. Dato che i compilatori possono svolgere analisi e ottimizzazioni statiche preliminari, i programmi compilati sono generalmente più rapidi dei programmi interpretati. D'altro canto richiedono, ad ogni modifica dei sorgenti, un nuovo ciclo di compilazione, che può richiedere molto tempo. I linguaggi interpretati, invece, possono essere eseguiti immediatamente, e sono generalmente molto più flessibili: è più facile per loro essere eseguiti interattivamente, agevolando così il debug e ogni genere di verifica a runtime. Questo rende Python un linguaggio ideale per creare applicazioni-prototipo, in modo rapido e dinamico.

1.1.4 Quindi i programmi scritti in Python sono 'lenti'?

Python è un linguaggio interpretato, ma adotta una serie di espedienti (spesso "presi a prestito" dal mondo della compilazione) per velocizzare le cose. Quando un modulo viene importato, ad esempio, viene creato un file "compilato" in una serie di istruzioni per l'interprete, dette **bytecode**. Queste istruzioni non sono rapide come quelle in linguaggio macchina, ma il sistema assicura che il modulo non debba essere riletto e interpretato inutilmente ogni volta, se non subisce cambiamenti. Alcune estensioni di Python, come Psyco, riescono a spingersi ancora più avanti, compilando porzioni di codice direttamente in linguaggio macchina, durante l'esecuzione del programma, raggiungendo in alcuni casi prestazioni simili a quelle dei linguaggi compilati. Ma in generale, nonostante tutti questi accorgimenti – che ne migliorano le prestazioni ad ogni nuova versione -, Python non può reggere il confronto in termini di rapidità di esecuzione con linguaggi di basso/medio livello, come assembly, C e C++. Questo, tuttavia, non è un problema. Nella pratica, le sezioni di un programma che hanno bisogno di prestazioni elevate sono sempre poche e circoscritte. In casi come questi, potete sfruttare le potenzialità di Python come **linguaggio collante**: se e quando avrete bisogno di una funzione ultrarapida, vi basterà scriverla in C/C++/PyRex/Altro e importarla direttamente nella vostra applicazione scritta in Python sotto forma di modulo di estensione.

1.1.5 L'interpretazione mi obbliga a rilasciare i miei sorgenti?

Potete scegliere: potete rilasciare i vostri sorgenti, anche assieme all'interprete, in modo da non obbligare l'utente a scaricarlo a parte. Oppure potete scegliere di non farlo: se non volete che gli altri vedano o possano modificare i vostri sorgenti, potete creare un eseguibile chiuso, proprio come fanno i linguaggi compilati. Nel gergo di Python, questi file vengono chiamati "**Frozen Binaries**" (file binari congelati). Alcuni programmi del tutto gratuiti, infatti, possono prendere il vostro codice sorgente, compilarlo in bytecode, impacchettarlo assieme all'interprete e mettere il tutto in un comodo file eseguibile (come un .exe su windows, o un .bin su linux). I "congelatori" più famosi, al momento, si chiamano pyInstaller, cx_Freeze, py2exe (solo per Windows) e py2app (solo per Mac OS X). Gli eseguibili risultanti non saranno di dimensioni ridotte come un semplice insieme di moduli, dal momento che dovranno contenere anche tutto l'interprete Python. Non saranno neanche più veloci, dal momento che non saranno compilati in linguaggio macchina, ma in bytecode, esattamente come farebbe l'interprete nel modo "tradizionale". Il vantaggio fondamentale è unicamente questo: saranno dei programmi stand-alone, autosufficienti, e gireranno senza problemi sulla/e piattaforma/e di destinazione come un qualsiasi eseguibile compilato.

1.1.6 Cosa significa 'multiparadigma'?

Nel corso degli anni sono stati sviluppati vari sistemi (detti più propriamente paradigmi) per strutturare un applicazione. Su queste basi teoriche i linguaggi costruiscono il proprio impianto logico e sintattico. Il linguaggio C, ad esempio, segue il paradigma procedurale, Prolog quello logico, Eiffel quello orientato agli oggetti. Nessuno di questi modelli è in assoluto migliore degli altri, ma ciascuno trova l'ambito in cui è più adatto. Per questo alcuni linguaggi, fra cui Python, hanno deciso di lasciare al programmatore la scelta su quale paradigma usare, adottandone più di uno.
Python permette di programmare in uno stile procedurale come il C, ma è anche fortemente orientato agli oggetti, tant'è vero che tutto in Python

è un oggetto. Inoltre, è possibile programmare con alcuni concetti presi a prestito da linguaggi che seguono il paradigma funzionale (come Lisp e Haskell), come le list comprehension, le funzioni lambda e map e l'applicazione parziale delle funzioni.
Riprenderemo bene tutti questi concetti nel corso del libro.

1.1.7 Che significa: 'a tipizzazione dinamica e forte'?

In molti linguaggi si usano degli **oggetti**, che appartengono naturalmente ad un preciso tipo (numeri, stringhe, file…). Un linguaggio di programmazione può richiedere che decidiate il tipo di ogni oggetto, nel momento stesso in cui scrivete il codice (in C, C++ e Java si fa così). Un linguaggio di questo tipo si dice **a tipizzazione statica**, ed è in grado di prevenire molti errori relativi a operazioni fra tipi incongruenti, prima ancora che parta il programma. Purtroppo questa sicurezza si paga in termini di flessibilità: dichiarare il tipo delle variabili implica la scrittura di più codice, e non sempre si può sapere in anticipo con quale tipo si avrà a che fare - molte volte un oggetto assumerà un tipo preciso soltanto nel corso dell'esecuzione e doverne anticipare uno per forza porta a serie limitazioni. Per questo, Python è tipizzato **dinamicamente**, ovverosia non è necessario indicare il tipo degli oggetti che usate mentre state scrivendo il codice. Gli oggetti in Python assumono un tipo soltanto dopo che sono stati creati, ed è solo da questo punto in poi che l'interprete inizia a controllare che non avvengano operazioni insensate fra tipi incongruenti (ad esempio: che si sommi un numero con una stringa). E' questo che alcuni affermano quando dicono che Python è **fortemente tipizzato**. (Non tutti, va detto, la pensano così. Purtroppo questo termine è molto ambiguo e troverete sempre esperti in disaccordo nel giurarvi che il linguaggio X sia debolmente o fortemente tipizzato, a seconda di cosa intendano precisamente con questi avverbi.). Secondo la definizione appena fornita, Python è senz'altro fortemente tipizzato: una volta che vengono creati, gli oggetti acquisiscono un tipo preciso e lo mantengono. A differenza di altri linguaggi a tipizzazione debole, in Python ci sono controlli molto rigidi

sulle operazioni ammissibili, è sempre possibile conoscere il tipo di una variabile (cosa che, ironicamente, a volte può non accadere in linguaggi a tipizzazione statica, come C e C++ - basti pensare ai puntatori void* e agli upcasting), e non può mai succedere che un oggetto cambi drasticamente il proprio tipo senza che il programmatore lo richieda esplicitamente. Questo tipo di tipizzazione, forte (le variabili hanno sempre un tipo significativo) e dinamica (questo tipo viene stabilito solo durante l'esecuzione), permette a Python di essere contemporaneamente robusto, flessibile e facile da usare. Inoltre è molto semplice in Python scrivere codice che analizzi e manipoli il programma stesso, rendendo naturali operazioni che in altri linguaggi si rivelano estremamente complesse e farraginose, come l'**introspezione** e la **metaprogrammazione**.

1.1.8 Che cos'è la 'libreria standard'?

La programmazione richiede l'uso degli strumenti più vari. Ecco, ad esempio, alcune operazioni basilari: leggere un file XML, mostrare a video una finestra, fare una ricerca testuale con una regex, serializzare i dati, generare numeri casuali, comunicare con un server, operare su thread distinti e interagire con i database. Tutte queste attività non possono essere previste direttamente nella sintassi del linguaggio, né possono essere reinventate ogni volta, pertanto vengono fornite dalle librerie. Molti linguaggi si affidano per la maggior parte di queste funzionalità a librerie esterne. Per realizzare una delle operazioni appena descritte in C++, ad esempio, dovrete andarvi a cercare di volta in volta gli strumenti più adatti messi a disposizione dalla comunità open source (boost, loki, wxWidgets…) o dal mercato. Ciò, soprattutto per i neofiti, è una grande seccatura e una fonte primaria di confusione. Per questo, Python integra assieme al linguaggio una ricca libreria standard, che contiene tutti gli strumenti per le esigenze medie del programmatore mainstream. È questo che s'intende comunemente col detto: "Python viene fornito a **batterie incluse**".

1.1.9 Python serve solo a scrivere script?

Python è un linguaggio fortemente dinamico e interattivo, quindi è par-

ticolarmente adatto alla realizzazione di script, un po' come Perl. Al contrario di quest'ultimo, però, Python può vantare una sintassi chiara e semplice, che rende il codice facile da leggere, scrivere e manutenere. Ma Python può essere usato per compiti ben più complessi della creazione di piccoli script: il suo modello a moduli e oggetti rende possibile la scrittura di intere applicazioni e framework. Il famoso client peer-to-peer Bittorrent, ad esempio, è stato scritto in Python, così come il noto application server Zope. Perfino alcuni videogiochi, come OpenRTS (che usa **pyGame**) e il MMORPG Eve-Online (i cui server usano una variante del linguaggio nota come stackless Python) sono realizzati in Python. Insomma: se volete scrivere un driver, o un compilatore veloce, rivolgetevi a linguaggi di medio/basso livello (personalmente vi consiglierei C e, soprattutto, C++). Per tutto il resto, potete considerare seriamente di scrivere almeno una buona parte della vostro progetto in Python. Un altro uso tipico di Python è la scrittura di **prototipi**: se dovete scrivere un'applicazione e non siete molto sicuri della sua struttura, provate a realizzarne un prototipo in Python: svilupperete in tempi rapidissimi e pian piano l'architettura si delineerà sotto i vostri occhi. Potrete provare varie alternative di sviluppo e vedere quale soddisfa meglio le vostre esigenze. Una volta creato il prototipo, potete riscriverlo in un linguaggio più rapido. Spesso vi accorgerete che non è necessario riscriverlo tutto: spesso è sufficiente limitarsi a trasformare le poche parti "sensibili" in estensioni C/C++.

1.1.10 D'accordo, voglio installarlo. Come faccio?

Per prima cosa dovete scegliere quale implementazione di Python usare. Quelle fondamentali sono almeno tre:

- **CPython**: È scritta in C, e per molte persone è, semplicemente, Python. In effetti, si tratta dell'implementazione "standard", quella che incorpora immediatamente i cambiamenti del linguaggio, e pertanto è sempre aggiornata.

- **Jython**: È scritta "al 100% in Java", e permette l'accesso a tutte le librerie di questo linguaggio. Le possibilità sono molto interessanti: ad esempio, potete usare Python per scrivere delle Java Applet!

- **IronPython**: È scritta in C#, e permette un'integrazione immediata fra le applicazioni scritte in .NET (e assembly della libreria) e Python.

Questo libro tratterà solo CPython, ma voi siete liberi di scaricare e provare anche le altre, se volete: coesistono tranquillamente sulla stessa macchina, senza alcun problema, e ciò che diremo qui – salvo funzionalità recenti ancora non supportate – sarà valido anche per le altre due.
Come ho già anticipato, se il vostro Sistema Operativo è basato su Linux o Mac OS, molto probabilmente l'ultima versione di Python è già installata è configurata correttamente sulla vostra macchina. Se state usando Windows, invece, dovrete installare CPython. Potreste farlo direttamente da http://python.org, ma vi consiglio, invece, di installare ActivePython. Si tratta di una distribuzione gratuita (ma non open-source) curata dalla compagnia ActiveState, che comprende, oltre a CPython, anche i moduli più usati (come ad esempio l'essenziale PyWin32, sotto windows), della documentazione aggiuntiva, e, soprattutto, la configurazione automatica dei percorsi e dell'interprete. Per installarlo, andate all'indirizzo http://activestate.com/activepython/, scegliete la distribuzione standard e scaricate l'ultima versione per il vostro Sistema Operativo. A questo punto non vi resta che eseguire il file binario e seguire il wizard d'installazione. Come mostra la figura 1.1, sotto Windows oltre alla documentazione e al classico interprete via shell, ActivePython fornisce anche un programma chiamato PythonWin. Si tratta di un ambiente di sviluppo integrato (scrit-
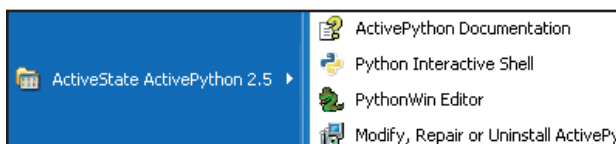
Figura 1.1: Esecuzione di una stored procedura che restituisce resultset

to in Python, ovviamente!), che rende più semplice e agevole la scrittura di applicazioni complesse, composte da un gran numero di moduli, funzioni e classi. Gli ambienti di sviluppo integrato sono delle gran comodità, e ci spenderete la maggior parte del tempo, pertanto vi converrà dare un'occhiata anche alle altre opzioni disponibili: io, ad esempio, mi trovo particolarmente a mio agio con PyScripter (http://mmm-experts.com/Products.aspx?ProductId=4). Altri ambienti interessanti sono l'immancabile **IDLE** (un buon editor cross-platform che viene fornito assieme a CPython e ad ActivePython, nella versione per Linux), **SPE, Eclipse** (con il plugin **PyDev**), **Boa Constructor**, **Wingware** e **Komodo**. Ma questa è solo una frazione delle possibilità offerte dal mercato: ogni giorno sembra nascere un nuovo IDE per Python.

1.2 … E UN BREVE GIRO DI PROVA

Ora che ci siamo procurati gli strumenti del mestiere, facciamo qualche piccolo esperimento per cercare di capire sommariamente come funzionano le cose. Si tratta di un'introduzione rapidissima e superficiale ad argomenti che serviranno da punto di partenza per i capitoli successivi, e nei quali, allo stesso tempo, verranno discussi ed nel dettaglio e approfonditi. Pertanto non è necessario che capiate ogni cosa nei particolari, ma solo che impariate i concetti fondamentali e il quadro d'insieme.

1.2.1 Usare l'interprete

Qualunque strumento abbiate installato, vi permetterà facilmente di interagire con l'interprete. Se non volete usare un IDE, potete semplicemente aprire la shell o il prompt dei comandi, e digitare il comando python. All'avvio dell'interprete vedrete qualcosa di simile:

```
Python 2.5.1 (r251:54863, May  2 2007, 16:56:35)

[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

La parte più importante sono quei tre simboli: “>>>”. Si tratta del prompt predefinito dell’interprete; in altre parole Python vi sta chiedendo gentilmente un’espressione da processare. Proviamo con qualcosa di semplice:

```
>>> 2 + 2
4
```

Su questo libro seguiremo sempre questa convenzione: una riga che inizia per “>>>” indica l’input scritto dall’utente, “…” indica la continuazione dell’input se questo si estende per più righe e il resto rappresenta la risposta dell’interprete. Quella che abbiamo appena scritto è un’**espressione numerica** (analizzeremo nel dettaglio questo tipo di espressioni nel prossimo capitolo). È anche possibile usare l’interprete per processare del testo, usando delle **stringhe**. Per ora, considerate semplicemente le stringhe come “del testo scritto fra un paio di apici o di virgolette”.

```
>>> “Ciao, mondo!”
‘Ciao, mondo!’
```

1.2.2 Creare ed eseguire script

A questo punto sappiamo usare l’interprete Python come una calcolatrice e come un merlo indiano. Per fare qualcosa di più interessante dobbiamo cominciare a creare dei piccoli script. Tecnicamente parlando, uno script è un semplice file di testo: pertanto se non volete usare un IDE, potete benissimo scrivere col semplice blocco note, o con vi. Un programma è una semplice sequenza di istruzioni. Esempio:

```
print “Ciao, Mondo!”
print “Stiamo scrivendo in Python!”
print “Bello, vero?”
```

Come vedete, non ci sono segni “>>>”, né risposte dell’interprete: adesso state programmando, e il lavoro dell’interprete non consisterà più nel calcolare espressioni, ma nell’eseguire le istruzioni,.una dopo l’altra. L’istruzione **print** ordina all’interprete di scrivere sull’output (solitamente la console) un certo valore (nel nostro caso, tre stringhe). Per eseguire questo programma senza usare un IDE, vi basterà salvarlo con estensione .py (ad esempio: ciaomondo.py), andare nella cartella in cui l’avete salvato, e scrivere al prompt dei comandi:

```
python ciaomondo.py
```

Python eseguirà il programma, restituendovi:

```
Ciao, Mondo!
Stiamo scrivendo in Python!
Bello, vero?
```

1.2.3 Usare le Variabili

Ora sappiamo anche scrivere degli script, ma sono ancora poco utili per l’utente. Per fare un passo avanti, dobbiamo introdurre il concetto di variabile. Detto in maniera semplice, una variabile è un nome assegnato ad un numero, ad una stringa, oppure ad altro (per dirla meglio: “un nome assegnato ad un **oggetto**”). Ad esempio, possiamo scrivere all’interprete:

```
>>> cambioEuroLire = 1936.27
```

Da questo momento fino alla chiusura dell’interprete (o fin quando non gli daremo un altro valore), cambioEuroLire sarà collegato a quest’espressione. Infatti, se chiediamo il valore di cambioEuroLire, l’interprete ci risponde prontamente:

```
>>> cambioEuroLire
1936,27
```

Esecuzione automatica

Su alcune versioni di windows, il sistema capisce automaticamente dall'estensione del file quale programma usare per aprirlo. Pertanto per eseguire il vostro script, vi basterà scrivere al prompt dei comandi:

ciaomondo.py

Per ottenere lo stesso risultato su unix/linux, si usa invece la tecnica classica dello shebang, indicando direttamente al sistema quale interprete usare per eseguire lo script. Vi basterà modificare ciaomondo.py, aggiungendo all'inizio del file la riga:

#!/usr/bin/env python

Possiamo usare cambioEuroLire proprio come useremmo 1936.27:

```
>>> soldiInEuro = 200
>>> soldiInLire = soldiInEuro * cambioEuroLire
>>> soldiInLire
387.254
```

1.2.4 Richiedere un input all'utente

L'ultima prova con l'interprete può darci lo spunto per realizzare un primo programma minimamente utile, per tutti coloro che ancora si ostinano a ragionare in lire: chiederemo all'utente una cifra in euro e gli restituiremo il corrispettivo in lire. Ma come si chiede un valore all'utente? Uno dei modi più semplici è usare la funzione raw_input("domanda"), che pone una domanda all'utente e restituisce la sua risposta sotto forma di stringa. Ecco input in azione, nello script EuroInLire.py:

```
cambioEuroLire = 1936.27
soldiInEuro = float(raw_input("Scrivi una cifra in euro: "))
```

```
soldiInLire = soldiInEuro * cambioEuroLire

print soldiInEuro, “euro equivalgono a”, soldiInLire, “lire”
```

Notate che nell’ultima istruzione ho usato la sintassi “print espr1, espr2,

Occhio alle maiuscole!

Piccolo test. Provate a scrivere questa riga:

```
>>> Print "Ciao!"
  File "<stdin>", line 1
    Print "Ciao!"
              ^
SyntaxError: invalid syntax
```

L’interprete vi segnalerà un bell’errore di sintassi. Perché? Perché avete scritto l’istruzione print con l’iniziale maiuscola.
Python è un linguaggio case sensitive, ovverosia considera diverse due parole uguali che cambiano soltanto per la disposizione delle maiuscole/minuscole.
Pertanto, fate molta attenzione, soprattutto ai nomi delle variabili: SOLDIINLIRE, SoldiInLire, soldiInLire e soldiinlire sono tutte variabili diverse e indipendenti l’una dall’altra!
Per chiarezza, quando dovrò definire nuovi nomi seguirò questa convenzione: le variabili inizieranno sempre con la lettera minuscola, classi, funzioni e metodi inizieranno sempre con la lettera maiuscola. In presenza di nomi composti indicherò l’inizio di ogni nuova parola con una maiuscola (ad esempio: numeroAureo, o StampaQuadrato).
Si tratta, come ho detto, di una convenzione, e non di una regola del linguaggio. Altri programmatori organizzano il loro codice diversamente, e anche Python usa un po’ un sistema misto per i nomi builtin e della libreria standard.

espr3, …" che permette di stampare più oggetti uno di seguito all'altro. Una possibile esecuzione sarà:

```
C:\> python EuroInLire.py
Scrivi una cifra in euro: 800
800 euro equivalgono a 1549016.0 lire
```

1.2.5 Commenti

Spesso nel corso del libro dovrò spiegare come funziona il codice, rivolgendomi a voi lettori dall'interno del codice stesso. Le annotazioni scritte per i lettori e non per l'interprete prendono il nome di **commenti**, e in Python vengono sempre precedute dal simbolo #. Un esempio di applicazione commentata può essere questo:

```
# Questo semplice programma chiede all'utente
# una temperatura in Farenheit e la restituisce in Celsius
# Chiedi all'utente la temperatura in F
tF = float(raw_input('Temperatura (in F)? '))
# Calcola la temperatura in celsius
tC = (tF-32) / 1.8
# Stampa i risultati
print tF, 'gradi Farenheit corrispondono a', tC, 'gradi Celsius'
```

Ogni volta che l'interprete incontra un commento, ignora tutto il resto della riga. Pertanto si possono scrivere anche commenti "a lato", come questo:

```
>>> print "Ciao!" #scrivi Ciao!
Ciao!
```

Saper commentare adeguatamente il codice è una delle abilità essenziali di un buon programmatore. I commenti che ho scritto in questo paragrafo, ad esempio, hanno senso qui solo perché siete inesperti e posso-

no esservi stati utili per capire come funziona l'applicazione. Un programmatore esperto, invece, li giudicherebbe pessimi: sono tutte informazioni inutili, facilmente desumibili dal codice stesso.
Via via che progrediremo nell'analisi del linguaggio impareremo a scrivere commenti più utili e significativi, e ad usare le docstring per creare applicazioni capaci di generare automaticamente documentazione per utenti e programmatori.

NUMERI ED ESPRESSIONI

Finora abbiamo usato i numeri senza entrare realmente nei dettagli. In questo capitolo scopriremo che in realtà Python mette a disposizione diversi tipi numerici: int, long, float e complex. Chiariremo anche il significato dei principali operatori in Python (cominciate a dare un'occhiata alla tabella §2.1), e impareremo a comporre, confrontare e analizzare espressioni numeriche e testuali.

2.1 TIPI NUMERICI

2.1.1 Int

Il tipo numerico più semplice in Python prende il nome di int. Chi conosce il C non si confonda: non c'entrano niente con gli int di quel linguaggio – in CPython sono, invece, modellati sul tipo long del C. Una costante di tipo int può essere scritta in molti modi: dal più semplice

```
>>> 1
1 #è un int!
```

Fino alle sue rappresentazioni in ottale (anteponendo uno '0' al numero), o esadecimale (anteponendo uno '0x' al numero).

```
>>> 64 # in decimale
64
>>> 0x40 #in esadecimale
64
>>> 0100 #in ottale
64
```

Notate che manca una delle basi fondamentali: il binario. Per rappresen-

tare un numero in binario (e in ogni altra base), potete richiamare il tipo int passando il numero sotto forma di stringa come primo argomento e la base come secondo argomento.

```
>>> int("1000000", 2) #in binario
64
```

Se il secondo argomento non viene specificato, viene sottinteso 10: in pratica si tratta di una conversione da stringa a intero. Ciò è molto utile perché, come abbiamo visto, Python non permette operazioni promiscue fra i tipi:

```
>>> 2 + '3'     #errore!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> 2 + int('3')     #convertiamo a intero
5 #OK!
```

2.1.2 long

A differenza delle comuni calcolatrici (e di molti linguaggi di programmazione), Python non pone limiti alla lunghezza dei numeri su cui potete operare.

```
>>> 2 ** 256   #cioè: 2 elevato alla 256
115792089237316195423
570985008687907853269984665640564039457584007913
                                      129639936L
```

In realtà i semplici int non consentirebbero di arrivare a queste dimensioni. Pertanto in questi casi Python usa un tipo di numero capace di espandersi per quanto serve. Questo tipo viene detto long, non c'entra nien-

te con il tipo long del C, e viene indicato con l'aggiunta di una 'L' alla fine del numero. Potete anche convertire un intero in long richiamando il tipo long(numero).

```
>>> 100L
100L
>>> long(100)
100L
```

Nella stragrande maggioranza dei casi, non dovrete preoccuparvi se un numero sia di tipo int o long: Python pensa da solo alle necessarie conversioni: un numero diventa automaticamente long quando esce dal raggio degli int, o quando un int viene combinato con un long.

```
>>> 100 - 1L
99L
```

2.1.3 float

Finora ci siamo limitati ai numeri interi, ma che dire di quelli con la virgola? Python permette di usarli nativamente grazie al tipo float (che corrisponde al tipo double, in C, e permette quindi la stessa precisione - sulla maggior parte dei PC moderni, 53 bit): basta che scriviate il numero usando il punto decimale. Potete anche usare un esponente per indicare la moltiplicazione per potenze di 10.

```
>>> 2e2
200.0
>>> 2e-2
0.02
```

Dietro le quinte, i numeri sono memorizzati col sistema della virgola fissa binaria. Se non avete mai avuto a che farci, andrete senz'altro incontro a delle sorprese. Anche se questa non è certo la sede per una spiega-

zione dettagliata della faccenda, è bene avvertirvi che certi numeri non sono direttamente rappresentabili. Ad esempio, 0.1:

```
>>> 0.1
0.10000000000000001
```

Quest'approssimazione suona un po' strana per l'utente. Se volete visualizzare un numero in modo più amichevole, potete convertirlo a stringa (vedremo cosa sono le stringhe più avanti), richiamando il tipo str, che ne restituisce una rappresentazione arrotondata.

```
>>> str(0.1)
'0.1'
```

Poiché l'istruzione print converte in stringa l'espressione da scrivere, l'output generato da uno script sarà già arrotondato:

```
>>> print 0.1
0.1
```

Anche per il numeri con la virgola vale il discorso della conversione automatica verso il tipo più complesso: combinare un intero con un float

Numeri Decimali

Nota: Il sistema binario si adatta perfettamente all'architettura del computer, ma per gli umani è molto più naturale usare il sistema della virgola mobile decimale. Se ne sentite la mancanza, la libreria standard di Python vi offre il tipo Decimal, che si trova nel modulo decimal.

```
>>> from decimal import Decimal
>>> Decimal('10') ** -1
Decimal('0.1')
```

genera un float.

```
>>> 1 + 2.0
'3.0'
>>> 1L + 2.0
'3.0'.
```

Se volete convertire nella direzione opposta (cioè passando dal tipo più complesso al più semplice) potete sempre richiamare il tipo più semplice, che provvederà al troncamento:

```
>>> int(3.8)
3
>>> long(10.8)
10L
```

2.1.4 Complex

Se avete esigenze matematico/ingegneristiche, sarete felici di sapere che Python dispone anche di un tipo builtin per i numeri complessi: complex. Un numero complesso può essere creato richiamando il tipo complex, indicando prima la parte reale e poi quella immaginaria:

```
>>> complex(1, 2)
1 + 2j
```

oppure, come mostra il risultato restituito dall'interprete, indicandone la componente immaginaria con la lettera j (indifferentemente minuscola o maiuscola):

```
>>> 1 + 2j
1 + 2j
```

Sui numeri complessi possono essere applicati tutti gli operatori già visti.

Ad esempio, possiamo calcolare le prime quattro iterazioni del di Mandelbrot per il punto (-0.2392, + 0.6507j):

```
>>> p = -0.2392 + 0.6507j
>>> p
(-0.2392+0.65069999999999995j)
>>> p + _**2
(-0.60539384999999999+0.33940511999999995j)
>>> p + _**2
(0.012105878135608011+0.23975245538697609j)
>>> p + _**2
(-0.29653468757864976+0.656504828015255j)
```

Potete accedere alla parte immaginaria o a quella reale di un numero complesso, attraverso i campi real e imag, del numero.

```
>>> p = -0.2392 + 0.65070j
>>> p.real
-0.2392
>>> p.imag
0.65069999999999995
```

Questi attributi sono a sola lettura, pertanto non possono essere usati per cambiare valore ad un numero complesso, pena il sollevamento di un eccezione (come vedremo nel prossimo capitolo, infatti, i numeri so-

Il segnaposto '_'

Notate che per riprodurre la ricorsone nella funzione di Mandelbrot senza usare una variabile, ho sfruttato un'interessante funzionalità dell'interprete: **il segnaposto** *"_" può essere utilizzato quante volte si vuole all'interno di un'espressione e* **rappresenta il valore dell'ultima risposta dell'interprete**.

no sempre immutabili).

```
>>> p.imag = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: readonly attribute
```

2.2 LE OPERAZIONI ARITMETICHE

Come abbiamo appena visto, Python può essere usato per la computazione di espressioni numeriche, tanto che alcune persone si limitano felicemente ad utilizzarlo come una calcolatrice molto potente e versatile. La tabella §2.1 mostra gli operatori aritmetici fondamentali:
Le operazioni aritmetiche sono, tutto sommato, molto semplici e intuitive. Qui di seguito sono presentati alcuni avvertimenti sui pochi elementi che potrebbero richiedere qualche attenzione particolare.

2.2.1 Precedenza, parentesi e leggibilità

E’ possibile combinare più operatori per la scrittura di espressioni composte:

Funzione	Uso	Significato	Precedenza
add	a + b	Addizione	3
sub	a – b	Sottrazione	3
mul	a * b	Moltiplicazione	2
div	a / b	Divisione	2
floordiv	a // b	Divisione intera	2
mod	a % b	Modulo (resto)	2
pow	a ** b	Elevamento a potenza	1
neg	-a	Negazione	1

Tabella 2.1: Operatori aritmetici fondamentali

```
>>> 10 / 2 + 3 * 5
20
```

Nella tabella §2.1 è indicato anche l'ordine di precedenza degli operatori (le operazioni col numero più alto hanno la precedenza). Nel caso si voglia alterarlo è sempre possibile usare le parentesi tonde.

```
>>> 10 / (2 + 3) * 5
10
```

E' possibile innestare un numero virtualmente illimitato di parentesi tonde, nei limiti del buon senso e della leggibilità.

```
>>> 4 ** (2 ** (4 / ((2 + 8) / 5)))
256
```

Le parentesi vengono normalmente usate dai programmatori anche per rendere più leggibili espressioni che potrebbero confondere altri lettori "umani":

```
>>> (10 / 2) + (3 * 5)
20
```

Tuttavia è meglio evitare di appesantire con parentesi inutili l'espressione. Quando è possibile, invece, è meglio giocare con gli spazi, che possono sempre essere aggiunti o rimossi a piacimento per rendere l'espressione il più leggibile e intuitiva possibile:

```
>>> 10/2 + 3*5
20
```

2.2.2 Divisione

Nel paragrafo sui decimali abbiamo visto che "combinare un intero con

Operatori e funzioni

Tutti gli operatori, in Python, vengono normalmente espressi con la notazione simbolica (come "a + b", o "–x"). Tuttavia ci sono casi in cui vi farà comodo considerarle delle come funzioni (come "add(a, b)", o neg(x)").

Le tabelle degli operatori qui presentate mostrano tutte, nella prima colonna, il nome della funzione corrispondente, così come è fornito dal modulo operator.

Qualche esempio:

```
>>> import operator
>>> operator.add(3, 2)
5
>>> operator.neg(10)
-10
```

Riprenderemo questa pratica quando parleremo della programmazione funzionale in Python.

un float genera un float". Ciò è interessante soprattutto nel caso della divisione. Nel caso in cui almeno uno dei termini di una divisione sia float, infatti, il risultato sarà un float (che terrà conto del resto).

```
>>> 44 / 6.0
7.333333333333333
```

Una divisione fra interi, invece, produrrà un intero (almeno, per adesso – vedi il box "la divisione nel futuro") che, ovviamente, non potrà tener conto del resto.

```
>>> 44 / 6
7
```

La divisione del futuro

A volte nel corso della storia di Python ci si accorge che sarebbe meglio cambiare certe cose per garantire una maggior coerenza del linguaggio. Il meccanismo della divisione è un buon esempio: il fatto che per gli interi funzioni diversamente che per i float è un'incoerenza e rende le cose inutilmente complicate. Pertanto si è deciso che nel futuro una divisione genererà sempre un float (che terrà conto del resto):

```
>>> 44 / 6
7 #nel presente; ma 7.333333333333333 in futuro
```

Questo comportamento è molto più coerente, e se serve una divisione intera sarà sempre possibile usare l'operatore //:

```
>>> 44 // 6
7
```

Se volete già cominciare ad usare questo sistema (e ve lo consiglio, perché prima o poi sarà quello utilizzato normalmente), potete ricorrere alla prassi che si usa in Python in questi casi: importare da __future__. Vedremo il meccanismo di import nel capitolo 5, nel frattempo immaginatevi l'istruzione "from __future__ import x" come "acquisisci magicamente la funzionalità x dal Python del futuro".

```
>>> from __future__ import division
>>> 44 / 6
7.333333333333333
```

Resto che, invece, potrà essere ottenuto tramite l'operatore modulo:

```
>>> 44 % 6
2
```

Talvolta, però, si vuole applicare questo tipo di divisione intera anche ai numeri float. In questo caso si può usare l'operatore //:

```
>>> 44 // 6.0
7.0
```

Ovviamente, dato che "la combinazione di numeri float genera sempre un float", il risultato sarà un float.

2.2.3 Operatori "in place"

Date un'occhiata a queste espressioni:

```
>>> a = a + 4
>>> a = a - 1
>>> a = a // 3
```

Assegnamenti del tipo "a = a op b" vengono comunemente definiti "in place", e capita piuttosto spesso di doverli scrivere. Per questo sono stati introdotti degli operatori particolari, mutuati direttamente dal C, che semplificano la scrittura in "a op= b".

```
>>> a += 2
>>> a -= 1
>>> a //= 2
```

Esiste un operatore in place per ciascun operatore aritmetico (e anche per quelli bit-a-bit che vedremo nel prossimo paragrafo). Il nome delle funzioni è lo stesso con l'aggiunta di una i iniziale (la versione in place di sub, ad esempio, è isub).

2.3 OPERATORI BIT-A-BIT

Come il linguaggio C, Python possiede degli operatori particolari, che

agiscono sui singoli bit di un intero. Questi operatori, riassunti in tabella §2.2, hanno una precedenza minore rispetto a quelli aritmetici e sono molto affini a quelli logici che vedremo nel paragrafo §2.5. And_, or_ e xor, operano sui bit di due interi, a coppie, e dispongono i bit risultanti in un intero. Esempi:

```
>>> 10 & 9
8
```

Il risultato è chiaro, se si tiene a mente che 10 in binario corrisponde a “1010”, 9 a “1001” e 8 a “1000”. La figura 2.1 mostra un esempio in colonna, per ciascun operatore.L’operatore ~ restituisce un intero in cui è stato applicato il complemento (ovvero in cui ogni bit è stato negato). Senza voler entrare nei dettagli del complemento a due (per i quali si rimanda ad

Funzione	Uso	Significato	Tabella di verità		
			a	b	Risultato
and_	a & b	And	0 0 1 1	0 1 0 1	0 0 0 1
or_	a | b	Or (o inclusivo)	0 0 1 1	0 1 0 1	0 1 1 1
xor	a ^ b	Xor (o esclusivo)	0 0 1 1	0 1 0 1	0 1 1 0
inv	~a	Complemento	0 1		1 0
rshift	a << n	Shift sinistro (sposta i bit di ‘a’ a sinistra di n posizioni)			
lshift	a >> n	Shift destro (sposta i bit di ‘a’ a destra di n posizioni)			

Tabella 2.2: Operatori bit-a-bit e relative tabelle di verità.

1010 &	1010 |	1010 ^
1001 =	1001 =	1001 =
1000	1011	0011

Figura 2.1: 10 & 9 = 8; 10 | 9 = 11; 10 ^ 9 = 3.

un qualsiasi testo decente di architettura dei computer), il risultato di ~x, sarà "–x –1". Pertanto, l'operatore complemento è sempre reversibile:

```
>>> ~10 #cioè (000000…1010)
-11 #cioè (111111…0101), o "-10 – 1"

>>> ~-11 #(111111…0101)
10 #cioè (000000…1010), o "-(-11) – 1"
```

Infine, gli operatori di lshift e rshift spostano verso sinistra o verso destra i bit di un intero, moltiplicandolo o dividendolo per potenze di due.

```
>>> int("110", 2) << 4
96 #int("1100000", 2).
```

Python è un linguaggio di alto livello, e normalmente non avrete alcun bisogno degli operatori bit-a-bit. Tuttavia se vi troverete ad usare librerie di basso livello per la manipolazione dei segnali, l'uso degli operatori &, |, << e >> vi permetterà di comporre o estrarre facilmente parti di un messaggio. Per fare un esempio: se state usando una libreria MIDI e dovete costruire il messaggio 0x90487F a partire dai tre byte costituenti: 0x90 ("suona sul canale 0"), 0x48 ("il Do centrale") e 0x7F ("con un'intensità di 127") - potete fare così:

```
>>> byte1 = 0x90
>>> byte2 = 0x48
>>> byte3 = 0x7F
```

```
>>> byte1<<16 | byte2<<8 | byte3
9455743 #cioè 0x90487F
```

2.4 OPERATORI RELAZIONALI

2.4.1 CONFRONTARE ESPRESSIONI

Una delle operazioni fondamentali in ogni linguaggio è il confronto fra due espressioni. In Python ciò è reso possibile dagli operatori relazionali elencati in tabella §2.3, che hanno una precedenza minore rispetto a quelli aritmetici e bit-a-bit. L'uso di questi operatori è del tutto intuitivo:

```
>>> print 1 <= 4, 4 <= 1
True False
```

In Python è possibile combinare più operatori relazionali uno di seguito all'altro, come se in mezzo ci fosse un operatore and. Questo concede di scrivere alcuni tipi di espressioni in modo molto più chiaro e sintetico rispetto a quanto avviene in molti altri linguaggi:

```
>>> x = 4
>>> 1 < x < 5  #in C: (1<x) && (x<5)
```

Funzione	Uso	Significato
eq	a == b	Uguale
ne	a != b	Diverso
Is_	a is b	Stessa istanza
is_not	a is not b	Diversa istanza
lt	a < b	Minore
le	a <= b	Minore o uguale
gt	a > b	Maggiore
ge	a >= b	Maggiore o uguale

Tabella 2.3: Gli operatori relazionali in Python

```
True
```

Oltre al sempre benvenuto zucchero sintattico, questa notazione ha anche dei vantaggi in termini di velocità di esecuzione e non presenta effetti collaterali, dal momento che ogni espressione viene valutata - al massimo - una volta sola.

2.4.2 BOOL

Come vedete dagli esempi, gli operatori relazionali restituiscono due costanti: True o False, per indicare l'esito del confronto. True e False sono due istanze del tipo bool (una sottoclasse di int) e corrispondono rispettivamente ai valori 1 e 0. La chiamata al tipo bool è particolarmente importante, dal momento che rivela se per Python l'argomento passato è valutato come True o come False. Per fare un esempio ovvio:

```
>>> bool(1)
True
>>> bool(0)
False
```

Python converte in False anche altri valori "nulli", come ad esempio il valore None, la stringa vuota (''), la lista vuota ([]) e la tupla vuota (()).

```
>>> bool("Stringa piena!")
True
>>> bool("")
False
```

2.5 OPERATORI LOGICI

Con gli operatori relazionali abbiamo imparato a scrivere singole proposizioni, che possono risultare vere o false. Gli operatori logici, mostrati in tabella §2.4, permettono inoltre di combinare più proposizioni, in ma-

niera concettualmente analoga agli operatori bit-a-bit. Hanno una precedenza inferiore a quella degli aritmetici, bit a bit e relazionali.

2.5.1 NOT, AND E OR

L'operatore not è simile al complemento: restituisce False se l'espressione è True, e viceversa:

```
>>> not True
False

>>> not 2 – 2
True #ricordate? bool(0) = False!
```

Tuttavia gli operatori and e or, in Python, hanno un comportamento peculiare. Superficialmente sono simili ai corrispondenti bit-a-bit & e |:

```
>>> True and False # 1 & 0
False #0

>>> True or False  # 1 | 0
True #1
```

Tuttavia la differenza emerge quando si confrontano espressioni non direttamente di tipo bool:

```
>>> 10 or 11
```

Uso	Restituisce
not a	True se a è False, altrimenti False
a or b	a se a è True, altrimenti b
a and b	a se a è False, altrimenti b
b if a else c	b se a è True, altrimenti c

Tabella 2.4: Operatori logici in Python

```
10
>>> 10 or 0
10
```

In entrambi i casi succede la stessa cosa: Python valuta il primo termine: bool(10) è True, pertanto è inutile valutare il secondo termine - l'espressione sarà True in ogni caso. Questo tipo di taglio (chiamato in gergo logica cortocircuitata) permette di risparmiare molto tempo quando le espressioni in gioco sono complesse, e viene usato da molti linguaggi di programmazione. Ma mentre altri linguaggi si limitano a restituire un generico True, Python restituisce l'ultimo valore che ha processato prima di fermarsi - in questo caso 10. Per capire il perché di questo comportamento, immaginatevi che succederebbe se i linguaggi fossero persone. Alla domanda: "Per andare a Piazza Navona devo girare a destra o a sinistra?", C++ e compagni risponderebbero "sì, è vero", e tirerebbero avanti ridacchiando come fanno gli hacker quando si sentono furbi e spiritosi. Python risponderebbe: "a sinistra". A rigor di logica entrambe le risposte sono corrette, ma quella di Python è l'unica effettivamente utile. Notate che il discorso vale anche se nessuna delle due espressioni è vera:

```
>>> 0 or ''
''
```

Qui Python prova a vedere quanto vale bool(0). È False, quindi prova con bool(''). È False, quindi si arrende e restituisce l'ultimo valore calcolato (''). La risposta è coerente proprio in virtù del fatto che il valore restituito è False, e False or False == False. Un discorso analogo vale per l'operatore and.

```
>>> 0 and 1
0

>>> 0 and ""
0
```

In entrambi i casi succede la stessa cosa: Python valuta il primo termine: bool(0) è False, pertanto è inutile valutare il secondo termine: l'espressione sarà False in ogni caso. Quindi Python restituisce l'ultimo valore analizzato (0). Se entrambi sono veri, invece:

```
>>> 1 and "Ciao!"
"Ciao!"
```

Python valuta il primo termine: bool(1) è True, quindi è costretto a valutare anche il secondo. bool("Ciao!") è True, quindi restituisce l'ultimo valore analizzato ("Ciao!"). Gli operatori logici possono essere concatenati. Provate, per esercizio, a computare mentalmente il valore di quest'espressione e il percorso esatto con cui Python ci arriva (e non barate usando subito l'interprete!):

```
>>> (0 and True) or ("Ciao" and "Mondo") or 1
```

2.5.2 L'OPERATORE TERNARIO

Python mette a disposizione anche un operatore ternario, capace di restituire un certo valore se una data condizione è vera, oppure un altro se è falsa. Un meccanismo del genere è presente in molti linguaggi come operatore (ad esempio il ? del C). La sintassi è questa:

```
(seVero if condizione else seFalso)
```

Ad esempio, se vogliamo definire una temperatura come "calda" se supera i 20 gradi, e "fredda" altrimenti, possiamo scrivere:

```
>>> temperatura = 40
>>> ('Calda' if temperatura > 20 else 'Fredda')

'Calda'
```

Fate attenzione all'ordine in cui Python valuta queste espressioni: prima condizione (temperatura > 20) e poi, a seconda del suo valore, seVero oppure seFalso (in questo caso "Calda", mentre "Fredda" non viene valutata). Come in tutti i linguaggi di programmazione, vale sempre il consiglio di non strafare con l'operatore ternario, dato che si tende facilmente a produrre codice incomprensibile. In caso di espressioni complesse è di gran lunga preferibile utilizzare le strutture di selezione, come spiegato nel paragrafo §4.1.

E xor?

Notate che al momento non esiste in Python un operatore xor logico, probabilmente per la difficoltà di definirne un comportamento coerente con questo sistema cortocircuitato.

Un'interpretazione possibile potrebbe essere quella di restituire il valore del termine True nel caso in ce ne sia uno solo e False altrimenti. In altre parole:

```
(not b and a) or (not a and b) or False
```

Per definire un buon surrogato di operatore xor con questo comportamento si potrebbe usare funzione lambda (vedi paragrafo §5.4.1), associata alla ricetta di Ferdinand Jamitzky per la creazione di funzioni a notazione infissa (vedi: http://aspn.activestate.com/ASPN /Cookbook/Python/Recipe/384122).

```
xor = Infix(lambda a, b: (not b and a) or (not a and b) or False)
```

Così diventa magicamente disponibile una funzione infissa "|xor|" col comportamento descritto.

```
>>>print 10 |xor| 20, 10 |xor| 0, 0 |xor| 20, 0 |xor| 0
False, 10, 20, False
```

2.6 FUNZIONI MATEMATICHE

Python offre un paio di funzioni builtin che hanno direttamente a che fare coi numeri: abs, per esempio, restituisce il valore assoluto di un espressione, anche di tipo complex:

```
>>> abs(-3+4j)
5.0
```

La funzione round arrotonda un float (può essere passato un secondo argomento che indica fino a quante cifre decimali arrotondare):

```
>>> print round(1.123), round(9.0 / 7.0, 3)
1.0 1.286
```

Le funzioni hex e oct restituiscono una stringa contenente la rappresentazione di un'espressione intera in esadecimale e ottale:

```
>>> print hex(127), oct(127)
0x7f 0177
```

Molte delle funzioni dedicate ai numeri, sono comunque contenute nei moduli math (per interi e float) e cmath (per i complex). Si tratta sostanzialmente di wrapper delle funzioni C presenti in <math.h>. Qualche esempio:

```
import math
import cmath
#trigonometria (pi, sin, atan, radians…)
>>> print math.sin(math.pi/2), math.atan(1) * 4
1.0 3.1415926535897931
>>> math.cos(math.radians(180))
-1.0
#radice quadrata, logaritmi, esponenziali
>>> math.sqrt(5)
```

```
 2.2360679774997898
>>> math.log(1)
0
>>> abs(cmath.exp(1j * math.pi))
1.0
#arrotondamenti (floor, ceil)
>>> math.floor(2.32)
2.0
>>> math.ceil(2.32)
3.0
```

2.7 ANDARE OLTRE

Gli strumenti numerici offerti da Python sono più che sufficienti per la maggior parte degli ambiti della programmazione. Se, lavorate nel settore scientifico, però, dovrete integrare qualche estensione. La più nota e vasta è SciPy, una vera e propria libreria di algoritmi, contenitori e strumenti matematici, che vanno dalle trasformate di Fourier e all'interpolazione, dalle funzioni per la statistica fino all'algebra lineare. SciPy è basato su un modulo a parte chiamato NumPy, che è valido di per sé, dal momento che permette di rappresentare e operare su matrici multidimensionali in modo efficace (anche il "vecchio" Numeric è ancora molto usato e apprezzato). Pygsl e Gmpy sono delle estensioni basate rispettivamente sulle note librerie GNU "gsl" e "gmp". Un numero imbarazzante di tool sono stati scritti per il plotting 2D (come Matplotlib) e 3D (come MayaVi).
Molti altri strumenti possono essere reperiti all'indirizzo: http://scipy.org/Topical_Software.

CONTENITORI

Nell'ultimo capitolo abbiamo discusso i tipi numerici, ma la programmazione non è fatta solo di numeri. È difficile trovare un'applicazione che non usi stringhe, liste e tabelle. Questi elementi hanno qualcosa in comune: sono contenitori di elementi: la stringa "ciao", ad esempio, può essere vista come un contenitore di singole lettere ('c', 'i', 'a', 'o'). In questo capitolo approfondiremo la nostra conoscenza dei principali contenitori offerti da Python: le sequenze (tuple, list e str), gli insiemi (set e frozenset) e le mappe (dict). Impareremo anche qualcosa in più su come Python gestisce le operazioni di assegnamento e di copia.

3.1 TUPLE

La sequenza più semplice in Python è il tipo tuple. Un oggetto di tipo tuple è una banale sequenza di n elementi di qualsiasi tipo.

3.1.1 Creare una tupla

Una tupla viene creata usando la seguente sintassi:

```
(elemento1, elemento2, elemento3…)
```

Ed ecco un esempio in pratica:

```
>>> (1, 2, 'CIAO!')
(1, 2, 'CIAO!')
```

Se una tupla contiene un solo elemento, c'è bisogno di usare una notazione particolare per far sì che Python la distingua da una semplice espressione numerica fra parentesi:

```
(elemento1, )
```

Ecco un esempio pratico:

```
>>> a = (“CIAO!”, )

>>> a
(“CIAO!”,)
```

In realtà nella definizione di una tupla le parentesi non sono sempre necessarie. L’esempio precedente, in effetti, si può anche scrivere così:

```
>>> 1, 2, ‘CIAO!’
(1, 2, ‘CIAO!’)
```

Python capisce da solo che si tratta di una tupla, perché in questo caso è evidente che non potrebbe trattarsi di altro. Tuttavia ci sono casi in cui la notazione senza parentesi è ambigua:

```
>>> print 1, 2, ‘CIAO!’
1, 2, ‘CIAO!’
```

In questo caso Python interpreta la sintassi come una chiamata alla versione di print con più argomenti. Non c’è modo di far capire che 1,2,”CIAO” è una tupla, se non mettendo gli elementi fra parentesi:

```
>>> print (1, 2, ‘CIAO!’)
(1, 2, ‘CIAO!’)
```

Una tupla può anche essere creata a partire da un qualsiasi oggetto iterabile (vedremo cosa significa esattamente questo termine, nel paragrafo §3.8), semplicemente passandolo come argomento di tuple.

```
>>> tuple(‘CIAO!’)
('C', 'I', 'A', 'O', '!')
```

3.1.2 Indentazione di più elementi

Se una tupla contiene molti elementi, scriverli tutti sulla stessa riga diventa impossibile (o quantomeno molto poco leggibile). Per questo quando si apre una parentesi (e solo allora), Python permette andare a capo e usare spazi e tabulazioni a piacimento per allineare gli elementi. Questa libertà finisce quando la parentesi viene chiusa.

```
>>> giorni = (
...     "lunedi'",
...     "martedi'",
...     "mercoledi'",
...     "giovedi'",
...     "venerdi'",
...     "sabato",
...     "domenica"
... )
```

Come vedete, in questo esempio l'interprete ha capito che l'istruzione non poteva terminare semplicemente dopo il primo "a capo", e ha atteso pazientemente che la parentesi venisse chiusa. I tre puntini di sospensione indicano, appunto, la continuazione dell'istruzione su più righe.

3.1.3 Funzioni builtin e Operatori

Per sapere quanti elementi sono presenti in una tupla, si può usare la funzione builtin len:

```
>>> len(giorni)
7
```

Le funzioni max e min, restituiscono il valore maggiore e minore in una sequenza:

```
>>> max(0, 4, 3, 1)
```

```
4
>>> min('Roberto', 'Allegra', 'Giabim')
'Allegra' #in ordine alfabetico
```

L'operatore + (come il rispettivo in place +=) può essere usato per concatenare più tuple.

```
>>> (1,3,5) + (2,4,6)
(1, 3, 5, 2, 4, 6)
>>> t = (1,"A",5)
>>> t += (2,"b",6)
>>> t
(1, 'A', 5, 2, 'b', 6)
```

L'operatore * (come il rispettivo in place *=) concatena più volte la stessa tupla.

```
>>> (1,2,3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> t = ('tanto',)
>>> t *= 5

>>> t
('tanto', 'tanto', 'tanto', 'tanto', 'tanto')
```

L'operatore in (detto di membership) verifica se un elemento è presente in una tupla.

```
>>> "lunedi'" in giorni, "apollodi'" in giorni
(True, False)
```

I comuni operatori relazionali funzionano anche per le tuple. Per la valutazione vengono confrontati i rispettivi elementi a coppie:

```
>>> (1,2,3,4) == (1,2,3,4)
True
>>> (1,2,3,4) < (1,2,3,3)
False
```

3.1.4 Accedere ad un elemento

Una volta creata una tupla, si può accedere ad uno degli elementi attraverso un indice numerico che parte da 0. Ecco un esempio pratico:

```
>>> numeri = ('zero', 'uno', 'due', 'tre')
>>> numeri[0]
'zero'
```

Se si accede ad un elemento che non esiste, viene lanciata una bella eccezione di tipo IndexError:

```
>>> numeri[4]
Traceback (most recent call last):

  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
Python permette anche di usare un indice negativo. Ad esempio

>>> numeri[-1]
'tre'
```

Questa scrittura è pari a:

```
>>> numeri[len(numeri) - 1]
'tre'
```

In pratica, gli indici negativi partono dall'ultimo elemento e vanno verso il primo, così come illustrato dalla figura 3.1

Figura 3.1: Gli elementi della tupla numeri, e i rispettivi indici positivi e negativi

3.1.5 Slice

Oltre ad un singolo elemento, è possibile ottenere una copia di una 'fetta' (in gergo: slice) di una tupla.

```
>>> numeri[1:3]
('uno', 'due')
```

La notazione n:m significa: "Parti dall'elemento n (incluso) e arriva fino all'elemento m (escluso)". Quindi in questo caso vengono presi gli elementi 1 e 2. Notate che l'indice di partenza è sempre minore (o uguale) a quello di arrivo. E questo è vero anche anche se passate come indici dei numeri negativi:

```
>>> numeri[-3:-1]
('uno', 'due')
```

Oltre al punto di partenza e quello di arrivo è anche possibile indicare un terzo argomento: il passo (o stride).

```
>>> numeriPari = numeri[0:4:2]

>>> numeriPari
('zero', 'due')
```

In questo caso è stato indicato uno slice dall'elemento 0 (incluso) all'elemento 4 (escluso), con uno stride di due (ovvero prendendo un numero sì e uno no).

In realtà questa scrittura può essere molto semplificata.

```
>>> numeriPari = numeri[::2]
>>> numeriPari
('zero', 'due')
```

Un indice di partenza vuoto, infatti, corrisponde al primo elemento, e un indice di arrivo vuoto corrisponde all'elemento… che segue l'ultimo! Un indice di stride vuoto corrisponde, infine, a 1.

```
>>> numeri == numeri[:] == numeri[::]
True
```

Anche l'indice di stride può essere negativo. In questo caso si conta a rovescio dall'indice di partenza a quello di arrivo.

```
>>> numeri[3:1:-1]
('tre', 'due')
```

Notate che se il passo è negativo, logicamente, un indice di partenza indica l'ultimo elemento e un indice di arrivo vuoto indica l'elemento… che precede il primo!

```
>>> numeriPariAlContrario = numeri[2::-2]
>>> numeriPariAlContrario
('due', 'zero')
```

3.1.6 Unpacking

Le tuple possono essere usate anche per assegnare più elementi a più variabili in con una sola istruzione (in gergo: effettuare un unpacking).

```
>>> a, b = 1, 2
>>> a
```

```
1
>>> b
2
```

Notate che l'unpacking è non è ambiguo, per cui le parentesi sono facoltative, e che la lunghezza della tupla di sinistra dev'essere pari a quella di destra. L'unpacking permette di scrivere in maniera semplice istruzioni più complesse. Un esempio tipico è lo scambio (o swapping) fra due o più variabili

```
>>> a, b, c = 'Q', 'K', 'A'  #assegnamento multiplo
>>> a, b, c = b, c, a        #scambio!

>>> a, b, c            #qual è l'asso?
('K', 'A', 'Q')
```

3.2 STRINGHE

Da un certo punto di vista una stringa può essere considerata una "sequenza di caratteri", e infatti tutte le operazioni e funzioni qui descritte per le tuple sono applicabili anche per le stringhe.
Il tipo str, però, espone molti metodi in più, che sono utili per la manipolazione dei caratteri. Questo paragrafo ne presenta alcuni.

3.2.1 Creare una stringa

Una stringa può essere espressa in più modi. Come abbiamo visto, il più semplice è racchiuderla fra virgolette o fra apici. Non c'è alcuna differenza fra le due notazioni, anche se di solito si preferisce quella fra apici. La notazione fra virgolette è comoda, però, se il testo stesso della stringa contiene degli apici:

```
>>> 'Ciao!', "lunedi'"
'Ciao', "lunedi'"
```

Come in C, le stringhe possono contenere delle sequenze di escape, che permettono "effetti speciali" al momento della stampa. Le più note sono \n (per andare a capo), \t (per inserire una tabulazione), e \', \" e \\ (per inserire rispettivamente un apice, una virgoletta e una backslash).

```
>>> print 'Ciao!\nMondo!'
Ciao!
Mondo!
```

Per scrivere del testo disabilitando le sequenze di escape, è sufficiente preporre alla stringa la lettera r (che sta per raw).

```
>>> print r'Ciao!\nMondo!\n'
Ciao!\nMondo!\n
```

Quando si distribuisce molto testo su più righe è fortemente consigliato usare le triple virgolette (" " "), che permettono di inserire a piacimento spazi e tabulazioni finché non vengono richiuse.

```
>>> print """Stiamo
... scrivendo
... su
... piu' righe"""
stiamo
scrivendo
su
piu' righe
```

Avrete sicuramente notato che, nel codice, finora ho usato gli apici al posto degli accenti ("piu'" al posto di "più"). Anche se questo è scorrettissimo in italiano, lo si accetta nel codice per il semplice motivo che le lettere accentate non fanno parte del codice ASCII di base. Se volete usare lettere e caratteri speciali, però, potete usare delle stringhe unicode, che

si indicano anteponendo alle virgolette la lettera u.

```
>>> print u"Così è più corretto!"
Così è più corretto!
```

Qualsiasi oggetto può inoltre essere convertito in una stringa. Come abbiamo già accennato, ad esempio, quelli di tipo numerico vengono convertiti in una rappresentazione testuale "leggibile":

```
>>> 1.1, str(1.1)
(1.1000000000000001, '1.1')
```

3.2.2 L'operatore %

Per comporre una stringa, si può usare il normale operatore di concatenamento che abbiamo già visto per le tuple. Quando la frase è lunga, però, la cosa può diventare un po' intricata:

```
>>> nome = raw_input("Come ti chiami? ")
Come ti chiami? Napoleone
>>> annoNascita = int(raw_input("In che anno sei nato? "))
In che anno sei nato? 1769
#Componiamo il saluto
>>> saluto = "Ciao, " + nome + "! Non li dimostri, i tuoi "
                              + str(2007 - annoNascita) + " anni!"
>>> saluto
'Ciao, Napoleone! Non li dimostri, i tuoi 238 anni!'
```

Come vedete, la composizione della frase è difficile da seguire, perché è lunga e, soprattutto, spezzettata. Eventuali dati non testuali (come l'età), inoltre, devono essere convertiti a stringa. E, per finire, si rischia di sbagliare spesso nel posizionare gli spazi all'inizio o alla fine del testo di raccordo. In questo caso, si può usare una stringa di formato, grazie all'operatore %:

```
>>> formato = "Ciao, %s! Non li dimostri, i tuoi %d anni!"
>>> saluto = formato % (nome, 2007 - annoNascita)
>>> saluto
'Ciao, Napoleone! Non li dimostri, i tuoi 238 anni!'
```

Questo sistema, molto più pulito, sarà sicuramente familiare a chi conosce la funzione printf del C. L'operando sinistro di '%' è una stringa (detta di formato) che contiene il testo da scrivere, all'interno del quale possono essere inseriti dei segnaposto. Questi vengono indicati con il simbolo % seguito dal tipo della variabile (%s per una stringa, %d per un intero, %f per un float, eccetera) e/o altre indicazioni posizionali o di formato (ad esempio, %.2f indica un float arrotondato alla seconda cifra decimale). L'operando destro di '%' è una tupla che contiene l'elenco delle variabili che verranno inserite, una dopo l'altra, nei rispettivi segnaposti.
L'operatore % semplifica la concatenazione delle stringhe, aiuta l'internazionalizzazione e la creazione dinamica di messaggi di testo. Per informazioni più dettagliate sulle possibilità di questo strumento e dei codici dei segnaposto, consultate la guida o un testo di riferimento.

3.2.3 Metodi delle stringhe

Il tipo str ha più di trenta metodi per analizzare il testo, o generarne una trasformazione. Quelli che seguono rappresentano soltanto una piccola selezione:

```
>>> # Maiuscole /Minuscole
>>> s = 'ciao, mondo!'
>>> s.lower(), s.capitalize(), s.title(), s.upper(),
'ciao, mondo', 'Ciao, mondo!', 'Ciao, Mondo!', 'CIAO, MONDO!',
>>> # Classificazione
>>>  'L'.isupper(), 'L'.islower()
(True, False)
>>> '2'.isalpha(), '2'.isdigit(),'2'.isalnum()
(False, True, True)
```

```
>>> # Eliminazione degli spazi
>>> s = "    CIAO, MONDO!    "
>>> s.lstrip(), s.rstrip(), s.strip()
('CIAO, MONDO    ', '    CIAO, MONDO', 'CIAO, MONDO')
>>> # Ricerca e sostituzione
>>> 'barbasso'.find('ba'), 'barbasso'.rfind('ba')
(0, 3)
>>> 'Vulneraria'.replace('aria', 'abile')
'Vulnerabile'
>>> ('dado' * 10).count('d')
20
>>> # Unione e divisione
>>> 'Beethoven,Bach,Bollani,Bettega'.split(",")
['Beethoven', 'Bach', 'Bollani', 'Bettega']
>>> ','.join(['Beethoven', 'Bach', 'Bollani', 'Bettega'])
'Beethoven,Bach,Bollani,Bettega'
```

3.3 NUMERI, STRINGHE E TUPLE SONO IMMUTABILI

A questo punto avrete notato che ho presentato una ricca disamina delle funzioni e delle possibilità di tuple e stringhe, ma non ho mai compiuto un'operazione banale: accedere ad un elemento e modificarlo.
Ad esempio:

```
>>> a = 'CIAO!'
>>> a[0] = 'M'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Come vedete dalla reazione dell'interprete c'è un motivo se ho evitato di farlo: tutti i tipi che abbiamo visto finora sono immutabili.

Figura 3.2: a fa riferimento a 123.456

Una volta creata una tupla, o una stringa, o anche un intero, non c'è verso di cambiarlo. Anche i metodi delle stringhe, in realtà, si limitano a creare una nuova stringa. Questo è un punto chiave di Python, che riesce paradossalmente più intuitivo ai novizi, piuttosto che agli esperti in altri linguaggi tipizzati staticamente, come C++ o Java. Pertanto vale la pena di approfondirlo.

3.3.1 Cosa succede in un assegnamento

Le variabili, in Python, sono solo dei nomi. Se siete dei novizi, potete vederli come delle frecce che indicano un certo oggetto. Se siete dei programmatori C o C++, potete vederle come dei puntatori.Quando scrivete:

```
>>> a = 123.456
```

In realtà state chiedendo a Python di fare due cose: creare l'oggetto float(123.456) e farlo puntare dalla variabile 'a' (vedi figura §3.2). D'ora in poi, a farà riferimento a quest'oggetto. Potete anche creare un'altra variabile, e assegnarla allo stesso oggetto:

```
>>> b = a
```

Ora sia 'a' che 'b' puntano effettivamente ad 123.456 (vedi figura §3.3). Potete avere conferma che si tratta dello stesso oggetto, usando l'operatore is:

```
>>> a is b
True
```

Figura 3.3: Sia a che b fanno riferimento a 123.456

Ora proviamo a riassegnare a.

```
>>> a = "CIAO!"
```

Alcuni programmatori che vengono da altri linguaggi a questo punto entrano in confusione, e cominciano a pensare che dato che sia a che b puntavano allo stesso oggetto, ora anche b dovrà valere "CIAO!". In
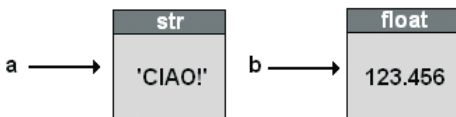
Figura 3.4: a fa riferimento a 'CIAO!', b a 123.456

realtà non è vero niente di tutto questo. È successa esattamente la stessa cosa che è successa la prima volta che abbiamo assegnato a. Python ha creato un nuovo oggetto ('CIAO!'), e l'ha fatto puntare dalla variabile a (vedi figura §3.4). A questo punto terminiamo il nostro esperimento riassegnando anche b.

```
>>> b = 10
```
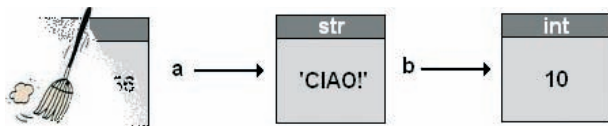
Figura 3.5: a fa riferimento a 'CIAO!', b a 10 e 123.456 viene rimosso dal Garbage Collection

Ora a punta a 'CIAO!' e b punta a 10. E l'oggetto 123.456? Che fine ha fatto? Python si accorgerà da solo che non è più referenziato da nessuno, e il garbage collector provvederà automaticamente a rimuoverlo dalla memoria. (vedi figura §3.5) Notate: coi nostri assegnamenti siamo stati soltanto in grado di creare nuovi oggetti, assegnargli nuovi nomi, o riassegnare quelli esistenti. Ma in nessun caso siamo riusciti a modificare minimamente il valore di un oggetto. Tutti i tipi visti finora, infatti, sono immutabili.

3.3.2 Gerarchie immutabili

Diamo un'occhiata più da vicino anche a come le sequenze come str e tuple vengono gestite in memoria. Cominciamo a creare una variabile a, assegnandole una tupla.

```
>>> a = (10,20,30)
```

Questa riga, in realtà, crea una serie di oggetti: i tre interi (di tipo int), e la tupla, che contiene tre riferimenti a loro collegati (vedi figura §3.6).
Grazie alle tuple possiamo creare facilmente delle vere e proprie gerarchie di oggetti che puntano ad altri oggetti. Per esempio, possiamo creare una tupla b che contiene a sua volta dei riferimenti alla tupla puntata da a:

```
>>> b = (a, 40, a)
>>> b
```
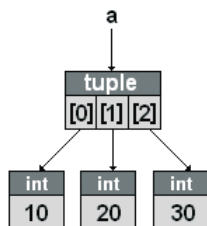
Figura 3.6: La struttura della tupla (10,20,30)

```
((10, 20, 30), 40, (10, 20, 30))
```

La figura §3.7 aiuterà a chiarire la situazione. Come vedete, le gerarchie possono diventare decisamente complicate. Notate anche che il tutto avviene per riferimento: se la tupla puntata da a cambiasse in qualche modo (diventando, ad esempio, (10, 20, "ciao mondo")), allora anche il valore della tupla puntata da b cambierebbe (diventando: (10, 20, "ciao mondo"), 40, (10, 20, "ciao mondo")). Ma le tuple sono immutabili, e questo ci garantisce che anche la nostra gerarchia non potrà che rimanere così.

3.4 LISTE
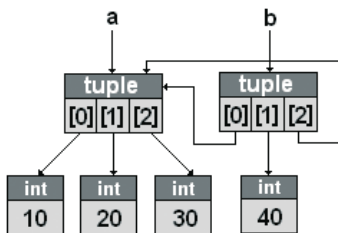
Figura 3.7: La struttura della tupla (a,40,a)

Le liste sono un altro tipo di sequenza, pertanto presentano delle profonde similitudini con stringhe e tuple. È possibile utilizzare sulle liste tutte le operazioni già viste per le tuple (membership, slice, concatenazione, moltiplicazione, eccetera…) Tuttavia le liste presentano una caratteristica che le rende uniche: sono mutabili. È possibile, in altre parole, assegnare dei valori ad un indice o ad una slice di una lista, alterandola. Questo apre una serie di nuove possibilità, metodi e operazioni. E, ovviamente, anche di problemi.

3.4.1 Creare una lista

Le liste appartengono al tipo list, e sono create secondo questa sintassi:

```
[elemento1, elemento2, elemento3…]
```

Un esempio immediato è:

```
>>> ['Berlioz', 'Boccherini', 'Brahms', 'Beccalossi']
['Berlioz', 'Boccherini', 'Brahms', 'Beccalossi']
```

Al solito, una lista può essere ottenuta partendo da qualsiasi oggetto iterabile, semplicemente passandolo come argomento di list. Ad esempio:

```
>>> list('Ciao')
['C', 'i', 'a', 'o']
```

La funzione builtin range restituisce la lista di tutti compresi fra il primo parametro e il secondo (escluso).

```
>>> range(-1, 5)
[-1, 2, 3, 4]
```

Si può anche usare un terzo parametro per indicare il passo (simile allo stride delle slice).

```
>>> range(1, 29, 2)

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27]
```

3.4.2 Assegnamento

Come dicevamo, gli elementi di una lista sono direttamente modificabili.

```
>>> linguaggi = ['C', 'C++', 'Perl']
```

Se vogliamo sostituire 'Perl' con 'Python' (e quasi sempre la cosa mi trova d'accordo), basta accedere all'elemento [2] e modificarlo:

```
>>> linguaggi[2] = 'Python'
```

```
>>> linguaggi
['C', 'C++', 'Python']
```

Possiamo anche assegnare un elemento (in questo caso dev'essere una lista) ad un'intera slice, sostituendo così un pezzo di lista con un'altra. Ad esempio, se abbiamo tendenze masochistiche, possiamo prendere 'C' e 'C++' e sostituirli con 'Java', 'C#' e 'Cobol':

```
>>> linguaggi[0:2] = ['Java', 'C#', 'COBOL']
>>> linguaggi
['Java', 'C#', 'COBOL', 'Python']
```

3.4.3 Aggiungere elementi

Dato che le liste sono mutabili, è possibile aggiungere elementi in place. Il metodo append aggiunge un elemento in coda:

```
>>> oggetti = ['tavolo', 'legno', 'albero']
>>> oggetti.append('seme')
>>> oggetti
['tavolo', 'legno', 'albero', 'seme']
```

Se invece di un solo elemento se ne hanno diversi in una qualunque sequenza iterabile, si può usare il metodo extend. Per mostrare la differenza, proviamo a passargli ancora 'seme':

```
>>> oggetti.append('seme')
>>> oggetti
['tavolo', 'legno', 'albero', 'seme', 's', 'e', 'm', 'e']
```

Se si vuole inserire un oggetto in una data posizione, si può usare il metodo insert:

```
>>> oggetti.insert(4, 'frutto')
>>> oggetti
```

```
['tavolo', 'legno', 'albero', 'seme', 'frutto', 's', 'e', 'm', 'e']
```

3.4.4 Rimuovere elementi

remove permette di rimuovere il primo elemento col valore specificato:

```
>>> >>> pezzi = ['Pedone', 'Cavallo', 'Alfiere', 'Torre', 'Regina', 'Re']
>>> pezzi.remove('Cavallo')
>>> pezzi
['Pedone', 'Alfiere', 'Torre', 'Regina', 'Re']
```

Se si vuole rimuovere un elemento conoscendone l'indice si può usare il metodo pop, che restituisce anche il valore dell'elemento cancellato:

```
>>> pezzi.pop(0)
'Pedone'
>>> pezzi
['Alfiere', 'Torre', 'Regina', 'Re']
```

Se il valore restituito non serve, è più semplice usare l'istruzione del:

```
>>> del pezzi[0]
>>> pezzi
['Torre', 'Regina', 'Re']
```

del può cancellare anche un'intera slice. Rimuoviamo i primi due elementi:

```
>>> del pezzi[:2]
>>> pezzi
['Re'] #Scacco matto!
```

3.4.5 Riarrangiare gli elementi

Il metodo reverse può essere usato per rovesciare una lista in place:

```
>>> figure = ['Fante', 'Cavallo', 'Re']
```

```
>>> figure.reverse()
>>> figure
['Re', 'Cavallo', 'Fante']
```

Analogamente, sort può essere usato per ordinare una lista in place:

```
>>> figure.sort()
>>> figure
['Cavallo', 'Fante', ‘Re’]
```

3.5 LE LISTE SONO MUTABILI!

Nel paragrafo §3.3 abbiamo visto nel dettaglio a cosa porta l’immutabilità di numeri, stringhe e tuple. È il caso di fare una serie di considerazioni analoghe anche per la mutabilità delle liste.

3.5.1 Copia superficiale

Cominciamo ad assegnare alla variabile a una semplice lista:

```
>>> a = [10, 20, 30]
```

ora puntiamo una variabile b sullo stesso oggetto di a:

```
>>> b = a
```

E ora viene il bello. Togliamo un elemento da a:

```
>>> del a[1]
```

Poiché b punta allo stesso oggetto di a, ora anche ‘b’ sarà cambiato:

```
>>> b
[10, 30]
```

Se questo vi ha sorpreso, non avete ancora afferrato bene cosa sono le variabili in Python (nomi: soltanto nomi senza tipo. O, se preferite: Frecce. Etichette. Alias. Riferimenti. Puntatori.). Se è il caso, provate a rileggere bene il paragrafo §3.3. Questo comportamento è spesso ciò che si desidera, ma a volte si vuole lavorare su due copie distinte della lista. Quindi, invece dell'assegnamento diretto (a = b) si dovrà creare una copia.Il metodo più semplice per copiare una lista è passarla come argomento di list:

```
>>> a = [10, 20, 30]
>>> b = list(a)
```

Ora, per la prima volta, vi trovate di fronte a due oggetti che hanno lo stesso valore, ma sono sicuramente distinti. Potete accorgervene usando gli operatori == e is.

```
>>> a == b,   a is b
(True, False)
```

Adesso non c'è più pericolo che un cambiamento di a si rifletta in b.

```
>>> del a[1]
>>> b
[10, 20, 30]
```

Un'alternativa un po' più elaborata è usare la funzione copy del modulo copy.

```
import copy

>>> a = [10, 20, 30]
>>> b = copy.copy(a) #come b = list(a)
```

Del

Del viene usata, in generale, per eliminare un riferimento (un elemento di un contenitore, una variabile, o un attributo). Ad esempio, proviamo a cancellare una variabile:

```
>>> a = 10
>>> del a
```

Da qui in poi sarà come se a non fosse mai stata assegnata.

```
>>> print a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

3.5.2 Copia profonda

La semplice copia via chiamata a list o copy.copy va sempre bene finché ci si limita a liste semplici. Come abbiamo visto, però, Python permette di innestare sequenze l'una dentro l'altra. Questo può creare problemi inaspettati.
Ci troviamo davanti ad una lista di scienziati del passato. Ogni elemento della lista è in realtà a sua volta una lista che contiene nome e cognome.

```
>>> a = [['Alan', 'Turing'], ['Jack', 'Von Neumann']]
```

Ci piace, e decidiamo di copiarla. Non vogliamo in alcun modo modificare l'originale, pertanto usiamo un sistema di copia.

```
>>> import copy
>>> b = copy.copy(a)
```

A questo punto ci accorgiamo che chi ha compilato la lista di nomi è un

po’ distratto, oppure un po’ ignorante: von Neumann si chiamava ‘John’, non ‘Jack’. Correggiamo:

```
>>> b[1][0] = 'John'
>>> b
[['Alan', 'Turing'], ['John', 'Von Neumann']]
```

Stiamo lavorando su una copia, quindi siamo sicuri che l’originale sarà rimasto intatto. Oppure no? Proviamo a vedere per sicurezza…

```
>>> a
[['Alan', 'Turing'], ['John', 'Von Neumann']]
```

Perché è cambiato anche l’originale? La figura §3.8 ci mostra il motivo: la copia che abbiamo eseguito ha duplicato soltanto la lista esterna ma non le sottoliste. La lista b è fisicamente distinta dalla a, ma i suoi elementi fanno riferimento agli stessi di a.L’operazione compiuta da copy.copy, o da list() si dice copia superficiale (o shallow copy): non si preoccupa di duplicare ricorsivamente tutti i suoi elementi. Per questo il modulo copy offre anche la funzione deepcopy, che effettua una copia profonda.

```
>>> a = [['Alan', 'Turing'], ['Jack', 'Von Neumann']]
>>> import copy
>>> b = copy.deepcopy(a)
```
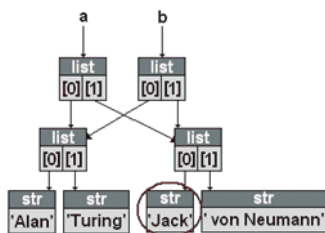
Figura 3.8: b è una copia superficiale di a

```
>>> b[1][0] = 'John'
>>> b
[['Alan', 'Turing'], ['John', 'Von Neumann']]
>>> a
[['Alan', 'Turing'], ['Jack', 'Von Neumann']]
```

Essendo per natura un'operazione ricorsiva, la copia profonda può sprecare molte risorse in termini di tempo e memoria. Fortunatamente i casi in cui si presenta davvero la necessità di copiare di peso tutti i sottoelementi di una sequenza sono piuttosto rari.

3.5.3 Gerarchie ricorsive

Vale la pena di accennare, fra le anomalie cui si può andare incontro giocando con le liste, alla possibilità di creare una struttura ricorsiva.

```
>>> io = ["io sono"]
>>> io.append(io)
['io sono', <Recursion on list with id=31785328>]
```

Python permette questo genere di strutture e riesce a riconoscere che il secondo elemento di questa lista si riferisce alla lista stessa. Pertanto lo segnala adeguatamente. Non c'è alcun problema ad accedere all'elemento io[1]: sarà uguale a io. E così via all'infinito.

```
>>> io[1][1][1][1][1][1]
['io sono', <Recursion on list with id=31785328>]
```

Se giocando con le liste il computer vi si impianta in un ciclo infinito, c'è la possibilità che abbiate involontariamente creato una struttura del genere.

3.6 DIZIONARI

I dizionari sono strutture molto flessibili e potenti. Si tratta sostanzial-

mente della versione che offre Python delle tabelle di hash. Sono contenitori iterabili, mutabili, di elementi non ordinati ai quali si accede attraverso una chiave.

3.6.1 Creare un dizionario

Il modo più semplice per creare un dizionario è racchiudere una serie di coppie fra parentesi graffe, secondo la seguente sintassi:

```
{chiave1: elemento1, chiave2: elemento2, …}
```

Esempio:

```
>>> ordine = {'coniglio': 'lagomorfi', 'topo': 'roditori', 'uomo': 'primati'}
```

Un altro sistema è richiamare il tipo dict, passando come argomento una sequenza contenente una serie di coppie "chiavi / valori". È particolarmente utilizzata in congiunzione con la funzione zip (per un esempio di costruzione con questo sistema, vedi il paragrafo §5.4.3). Infine, un sistema pratico consiste nel creare un dizionario vuoto:

```
>>> ordine = {} #o ordine = dict()
```

e popolarlo via via assegnando direttamente via indice (nel prossimo paragrafo vedremo come).

3.6.2 Accesso agli elementi e assegnamento

La particolarità che rende tanto utili i dizionari è l'accesso agli elementi e la loro creazione. Nei contenitori visti finora si è utilizzato l'accesso via indice, ma i dizionari non sono tipi ordinati. Ad un dizionario si accede via chiave. Ecco, ad esempio, come recuperare l'ordine dei conigli:

```
>>> ordine['coniglio']
```

```
'lagomorfi'
```

Grazie alla rappresentazione via hash, questa scrittura ha sostanzialmente la stessa rapidità di un accesso via indice: il valore viene recuperato direttamente a partire dalla chiave. Lo stesso sistema è anche usato per la modifica di un elemento esistente. Ma soprattutto, per la prima volta, possiamo usare l'assegnamento via indice per creare un elemento nuovo.

```
>>> ordine[volpe'] = 'canidi'
>>> ordine
{'coniglio': 'lagomorfi',
 'topo': 'roditori',
 'uomo': 'primati',
 'volpe': 'canidi'}
```

L'accesso in lettura ad un elemento non esistente causa, invece, il sollevamento di un'eccezione:

```
>>> ordine['tasso']
Traceback (most recent call last):

  File "<stdin>", line 1, in <module>
KeyError: 'tasso'
```

Per controllare preventivamente se una certa chiave esiste si può ricorrere al classico operatore di membership, oppure alla funzione has_key:

```
>>> 'tasso' in ordine, ordine.has_key('tasso')
(False, False)
```

Il metodo get è analogo alla lettura via indice, ma in caso di fallimento non solleva un eccezione – si limita a restituire il tipo None, oppure il valore passato come secondo argomento:

```
>>> ordine.get('coniglio'), ordine.get('tasso', 'Non pervenuto')
('lagomorfi', 'Non pervenuto')
```

I metodi keys e values tornano utili molto spesso: restituiscono rispettivamente la lista delle chiavi e quella dei valori di un dizionario:

```
>>> ordine.keys(), ordine.values()
(['coniglio', 'topo', 'volpe', 'uomo'],
 ['lagomorfi', 'roditori', 'canidi', 'primati'])
```

3.6.3 Usi 'furbi' dei dizionari

Programmando in Python si usano i dizionari in una serie sorprendente di contesti. Si va dal 'normale' uso come tabella di hash ad una serie di idiomi ormai noti. Usando le tuple come chiave, ad esempio, è possibile simulare una matrice sparsa:

```
>>> nodo = dict()
>>> nodo[(200, 4, 120)] = 'Corazzata'
>>> nodo[(100, 240, 0)] = 'Incrociatore'
>>> #l'avversario tenta di colpire
>>> tentativo = (200,3,120)
>>> #vediamo se mi ha colpito
>>> nodo.get(tentativo, 'Acqua!')
'Acqua!'
```

Nell'esempio qui sopra, mi è bastato usare due tuple per simulare un campo di gioco per battaglia navale di almeno 200 * 240 * 120 nodi. (A proposito, non mi stupirei se la partita durasse qualche annetto). Notate l'uso del secondo argomento nel metodo get per fingere che tutti i nodi non presenti abbiano effettivamente il valore 'Acqua!'. In modo del tutto analogo, usando una chiave numerica, si può simulare una lista infinita:

```
>>> lista = dict()
```

```
>>> lista[200] = '20'
>>> lista[300] = '10'
>>> lista[2**300] = '20'
```

Come vedremo nel capitolo 4.4, un altro uso tipico dei dizionari consiste nella simulazione del meccanismo di switch tipico del C.

3.7 INSIEMI

Python fornisce due tipi, set e frozenset, per la gestione di insiemi (rispettivamente mutabili e immutabili). Gli insiemi sono visti come degli oggetti iterabili, che contengono una sequenza non ordinata di elementi.

3.7.1 Creare un insieme

Un insieme viene creato richiamando il tipo set (o frozenset) passando come argomento una qualsiasi sequenza iterabile:

```
>>> numeriDispari = set([1, 3, 5, 7, 9, 11, 13, 15, 17])
>>> numeriPrimi = set([1, 3, 5, 7, 11, 13, 17])
```

3.7.2 Operazioni sugli insiemi

Gli insiemi sono iterabili, pertanto è possibile effettuare tutte le operazioni di base (membership, min, max, len, eccetera).

```
>>> 15 in numeridispari, 15 in numeriprimi
(True, False)
```

I metodi caratteristici di set e frozenset corrispondono a quelli classici dell'insiemistica:

```
>>> numeriDispari.difference(numeriPrimi)
set([9, 15])
>>> numeriDispari.intersection(numeriPrimi)
```

```
set([1, 3, 5, 7, 11, 13, 17])
>>> numeriprimi.issubset(numeridispari)
True
```

Molti di questi metodi hanno un corrispettivo operatore, ottenuto sovraccaricando il significato di quelli aritmetici o bit-a-bit:

```
>>> numeridispari - numeriprimi
set([9, 15])
>>> numeriDispari & numeriprimi
set([1, 3, 5, 7, 11, 13, 17])
```

Se l'insieme è mutabile, è possibile usare una serie di metodi per aggiungere o rimuovere elementi:

```
>>> numeridispari.add(19)
>>> numeriprimi.discard(1)
>>> numeriprimi, numeridispari
([3, 5, 7, 11, 13, 17], [1, 3, 5, 7, 9, 11, 13, 15, 17, 19])
```


3.8 ITERATORI

Tutti i contenitori che abbiamo visto sinora sono iterabili. Questo vuol dire che sono in grado di generare un iteratore: un oggetto capace di percorrerne gli elementi.

3.8.1 Creare un iteratore

A differenza dei contenitori, un iteratore non appartiene ad un tipo univoco. Qualsiasi classe può essere un iteratore a patto che implementi un protocollo adatto – cioè, fondamentalmente, disporre di un metodo next() che restituisca l'elemento successivo.
Per ottenere un iteratore adatto a sfogliare una certa sequenza, basta richiamare la funzione builtin iter:

```
>>> nipoti = ['Qui', 'Quo', 'Qua']
>>> iteratore = iter(nipoti)
>>> iteratore
<listiterator object at 0x03EC7C50>
>>> iteratore.next()
'Qui'
>>> iteratore.next()
'Quo'
>>> iteratore.next()
'Qua'
```

Se si richiama next quando gli elementi sono finiti viene sollevata un eccezione di tipo StopIteration:

```
>>> iteratore.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Questo sistema garantisce di poter "sfogliare" gli elementi di un contenitore senza doversi preoccupare di quale sia il suo tipo. Il linguaggio stesso sfrutta ampiamente gli iteratori dietro le scene, nella gestione dei cicli for (approfondiremo la cosa nel paragrafo §4.3.4). La funzione iter fa generare alla sequenza l'iteratore predefinito, ma alcuni tipi possono generare anche altri tipi di iteratori. I dizionari, ad esempio generano per default un iteratore sulle chiavi:

```
>>> capitali = {'Giappone': 'Tokio', 'Italia': 'Roma'}
>>> iteratoreChiavi = iter(capitali)
>>> #Generiamo una lista a partire dall'iteratore
>>> list(iteratoreChiavi)

['Italia', 'Giappone']
```

Questo tipo di iteratore può essere generato anche dal metodo dict.iterkeys(). Ma se vogliamo iterare sui valori (cioè, nell'esempio, le capitali) invece che sulle chiavi (gli Stati), possiamo usare dict.itervalues().

```
iteratoreValori = capitali.itervalues()
>>> #Generiamo una tupla a partire dall'iteratore
>>> tuple(iteratoreValori)
('Roma', 'Tokio')
```

Se vogliamo iterare su una serie di tuple (chiave, valore), possiamo anche usare dict.iteritems()

```
>>> iteratoreTupla = capitali.iteritems()
>>> iteratoreTupla.next()
('Italia', 'Roma')
>>> iteratoreTupla.next()
('Giappone', 'Tokio')
```

3.8.2 Altri tipi di iteratori

I contenitori non sono gli unici oggetti capaci di creare degli iteratori. Python fornisce alcune funzioni builtin a questo scopo, che si rivelano particolarmente utili nei cicli for. reversed, per esempio, prende come argomento una sequenza e restituisce un iteratore che parte dall'ultimo elemento e va verso il primo:

```
>>> tuple(reversed['zero', 'uno', 'due'])
('due', 'uno', 'zero')
```

enumerate prende come argomento un qualsiasi oggetto iterabile e restituisce ad ogni chiamata di next una tupla in cui all'elemento corrente viene affiancato il numero totale delle iterazioni effettuate.

```
>>> list(enumerate(['zero', 'uno', 'due']))
```

```
[(0, 'zero'), (1, 'uno'), (2, 'due')]
```

Altri iteratori possono essere costruiti facilmente attraverso gli strumenti offerti dal modulo itertools (consultate un manuale di riferimento per l'elenco completo):

```
>>> import itertools
>>> #otteniamo un bel ciclo infinito
>>> ciclo = itertools.cycle(('Casa', 'Dolce'))
>>> ciclo.next(), ciclo.next(), ciclo.next(), ciclo.next(), ciclo.next()
('Casa', 'Dolce', 'Casa', 'Dolce', 'Casa')
```

Potete anche costruire un iteratore da voi, definendo i metodi __iter__ e next. Oppure potete usare un generatore (vedi §5.4.7).

CONTROLLO DEL FLUSSO

Ora che abbiamo accumulato un po' di esperienza coi tipi di base è il momento di cominciare a prendere le redini della programmazione. Finora abbiamo giocato con l'interprete: ora dobbiamo concentrarci sulla creazione di script. Come abbiamo visto, questo significa che scriveremo serie di istruzioni che verranno eseguite una dopo l'altra.
Spesso questo flusso va alterato in qualche modo: a volte bisogna eseguire certe istruzioni solo se si verificano determinate condizioni (ad esempio, eseguire una divisione solo se il divisore è diverso da zero). Oppure bisogna ripetere un'istruzione n volte, o finché non si verifica un certo evento.

4.1 IF

Cominciamo proprio dall'esempio accennato nell'ultimo paragrafo: vogliamo scrivere un programma che esegua una divisione fra due numeri passati dall'utente.

```
#divisione1.py

a = float(raw_input('Digita il primo numero: '))
b = float(raw_input('Digita il secondo numero: '))

print 'Operazione effettuata:'

print '%f / %f = %f' % (a, b, a/b)
```

Il prgramma funziona bene, finché b è diverso da 0. In questo caso, invece, viene sollevata un'eccezione di tipo ZeroDivisionError:

```
C:\> python divisione1.py

Digita il primo numero: 18
Digita il secondo numero: 0
Traceback (most recent call last):
  File "divisione1.py", line 6, in <module>
    print '%f / %f = %f' % (a, b, a/b)
ZeroDivisionError: float division
```

Normalmente il comportamento più giusto sarebbe intercettare l'eccezione, ma ancora non lo sappiamo fare. Per ora ci limiteremo ad evitare di effettuare la divisione nel caso in cui b sia diverso da zero. Per questo dobbiamo usare il costrutto if, che nella sua forma più semplice ha la seguente sintassi:

```
if condizione:
    istruzioni
```

Ed ecco come diventa il nostro codice:

```
#divisione2.py
#… resto del programma

if b != 0: #o anche semplicemente: if b:
    print 'Operazione effettuata:'
    print '%f / %f = %f' % (a, b, a/b)
```

4.1.1 If … else

Ora l'applicazione non solleva più l'eccezione in caso di b==0, ma non avverte l'utente dello sbaglio! Per questo dobbiamo prevedere un caso alternativo, estendendo la sintassi di if.

```
if condizione:
```

```
    istruzioniSeVera
else:
    istruzioniSeFalsa
```

Indentazione

INotate che le istruzioni all'interno del blocco if sono precedute da una serie di spazi. Questa pratica viene chiamata indentazione, e viene seguita nella maggior parte dei linguaggi di programmazione. In questo modo si permette al lettore di riconoscere subito il livello di innestamento di un blocco rispetto alle istruzioni circostanti.

```
IstruzioneLivello1
IstruzioneLivello1:
    IstruzioneLivello2:
        IstruzioneLivello3
        IstruzioneLivello3
    IstruzioneLivello2
IstruzioneLivello1
```

Ma mentre in molti altri linguaggi indentare il codice è un'educata convenzione, in Python si tratta di una parte integrante della sintassi. Quando innestate un blocco di istruzioni dovete indentarle, separandole dal livello superiore con un certo numero di spazi bianchi (potete scegliere quanti e quali, purché li manteniate costanti. Molti scelgono 4 spazi, alcuni preferiscono 1 tabulazione - cosa che consiglio di evitare, se non volete problemi a trasportare il vostro codice in altri editor).
Quest'uso sintattico degli spazi bianchi appare sconcertante a chi ha sempre conosciuto soltanto linguaggi in cui i blocchi si aprono e si chiudono con delimitatori (come le parentesi graffe del C, o il BEGIN…END del Pascal). Dopo un primo attimo di disappunto, la maggior parte degli sviluppatori scopre che questo sistema porta alla scrittura di codice molto pulito e leggibile.

Ed ecco il nostro programma:

```
#divisione3.py
#… resto del programma …
if b != 0:
   print 'Operazione effettuata:'
   print '%f / %f = %f' % (a, b, a/b)
else:
   print 'Operazione non riuscita:'
   print 'Non so dividere per zero!'
```

Ora, quando b sarà != 0 il programma eseguirà le istruzioni contenute nel blocco if, (cioè la divisione). Altrimenti eseguirà le istruzioni in else.

```
C:\> python divisione3.py

Digita il primo numero: 18
Digita il secondo numero: 0
Operazione non riuscita:
Non so dividere per zero!
```

4.1.2 if … elif … else

Proviamo a creare un programma che, dati due interi a e b in ingresso dica se a > b, a < b, oppure se a == b.
#maggioreminore1.py

```
a = float(raw_input('Digita il primo numero: '))
b = float(raw_input('Digita il secondo numero: '))

if a > b:
   print "Il primo numero e' maggiore del secondo"
if a < b:
```

```
    print "Il primo numero e' minore del secondo"
if a == b:
    print "Il primo numero e' uguale al secondo"
```

Questa prima versione è semplice e leggibile. Però forza inutilmente l'interprete a compiere tre verifiche in ogni caso. Sfruttando la sintassi di if…else possiamo fare di meglio:

```
#maggioreminore2.py
#…resto del programma

if a > b:
    print "Il primo numero e' maggiore del secondo"
else:
    if a < b:
        print "Il primo numero e' minore del secondo"
    else:
        print "Il primo numero e' uguale al secondo"
```

Questa seconda versione è più ottimizzata. Se a è maggiore di b basterà un solo test. In caso contrario ne basteranno comunque solo due. Innestando siamo riusciti a scrivere tutto dentro un unico blocco if, invece di usarne tre separati.
Il problema è che questa scrittura è un po' meno leggibile di quella di prima (immaginatevi cosa succederebbe se i casi alternativi fossero sei o sette, invece di tre). Per questo Python prevede un ulteriore sintassi di if:

```
if condizione1:
elif condizione2:
elif condizione3:
…
else:
```

Ed ecco il nostro programma leggibile e ottimizzato:

```
#maggioreminore3.py
#…resto del programma

if a > b:
    print "Il primo numero e' maggiore del secondo"
elif a < b:
    print "Il primo numero e' minore del secondo"
else:
    print "Il primo numero e' uguale al secondo"
```

4.2 WHILE

Ora sappiamo far prendere direzioni diverse al flusso di esecuzione a seconda dei casi. Possiamo anche fare in modo che un blocco di istruzioni venga ripetuto fintantoché si verifica una certa condizione. Il costrutto while serve a questo, e nella sua forma più semplice ha la seguente sintassi:

```
while condizione:
    istruzioni
```

4.2.1 Ciclo infinito

Con while possiamo scrivere un programma che non termina mai:

```
#kamikaze.py

while True:
    print "BANZAI!"
```

Quello che abbiamo scritto è un ciclo infinito (True, infatti, è

sempre vero per definizione). L'unico modo per abbattere il kamikaze è spararagli premendo la combinazione di uscita (spesso CTRL+C o CTRL+Z, a seconda del sistema operativo).

```
c:\> python kamikaze.py
BANZAI!
BANZAI!
BANZAI!
BANZAI!
… (Basta! premo CTRL+C) …
Traceback (most recent call last):
  File "kamikaze.py", line 4, in <module>
    print "BANZAI!"
KeyboardInterrupt
```

4.2.2 Pass

Proviamo a scrivere un programma più educato, che chieda all'utente quando uscire:

```
#pass.py

ok = ('s', 'si', "si'", u'sì', 'y', 'ok', 'certo', 'basta!')

while raw_input("Adesso vuoi uscire? ").lower() not in ok:
    pass

print "\n\nArrivederci!"
```

Qui il ciclo continua finchè l'utente non scrive una risposta fra quelle contenute nella tupla ok.
Notate che il grosso del lavoro viene svolto nella condizione di uscita, nella quale si richiede la risposta e la si confronta: il corpo del ciclo è vuoto. Tuttavia Python richiede sempre che scrivia-

te almeno un'istruzione. In questi si usa l'istruzione pass, che non fa nulla e viene usata solo come "riempimento".

4.2.3 Continue

A volte in un ciclo bisogna saltare un'iterazione e passare direttamente alla successiva. Scriviamo un piccolo script che conti i primi cinque piani di un edificio:

```
piano = 0

while piano < 5:
    piano += 1
    print "Piano ", piano
```

Supponiamo di essere in un albergo giapponese, in cui il piano numero 4 (shi) non esiste per motivi scaramantici (shi vuol dire anche "morte", il che non è il massimo dell'ospitalità). Per saltare l'iterazione quando piano == 4 possiamo usare continue.

```
#continue.py

piano = 0

while piano < 20:
    piano += 1
    if piano == 4
        continue
    print "Piano ", piano
```

L'istruzione continue salta tutto il resto del blocco andando direttamente all'iterazione successiva.

```
C:\>python continue.py
```

```
Piano 1
Piano 2
Piano 3
Piano 5
```

Come dimostra questo stesso caso, spesso è molto più semplice e leggibile usare un semplicissimo blocco if al posto di continue:

```
if piano != 4:
    print "Piano ", piano
```

Tuttavia a volte quest'istruzione può essere utile per non creare innestamenti troppo profondi.

4.2.4 Break

Altre volte bisogna interrompere un ciclo se al suo interno si verifica una data condizione. In questo caso si può usare l'istruzione break. Se voglio sapere se un certo numero è primo o meno, scriverò qualcosa del genere:

```
#break.py

numero = int(raw_input("Scrivi un numero > 1: "))

fattore = numero / 2

while fattore > 1:
    if numero % fattore == 0:
        print "Non e' primo: si puo' dividere per", fattore
        break

    fattore -= 1
```

```
if fattore <= 1:
        print "E' primo!", fattore
```

All'interno del ciclo verifico per ogni fattore se questo è un divisore del numero passato dall'utente. In caso affermativo, è inutile continuare la computazione: è chiaro che non può essere primo, pertanto forzo l'uscita dal ciclo con l'istruzione break.

4.2.5 while ... else

Notate che nell'esempio precedente, subito dopo il ciclo, mi trovo in una strana situazione: non so se si è usciti dal while perché è stato trovato un fattore e quindi c'è stato un break, oppure se il ciclo è terminato naturalmente, perché fattore è <= 1. Per questo sono costretto a usare un blocco if.
Questa situazione è tipica, tanto che Python prevede un'ulteriore estensione della sintassi di while:

```
while condizione:
    istruzioni
else:
    istruzioni
```

Le istruzioni in else vengono eseguite soltanto se si è usciti dal ciclo perché la condizione non è più vera (nel nostro caso, perché fattore non è più > 1). Se, invece, il ciclo termina in modo "innaturale" (ad esempio, per un break), il blocco in else non viene eseguito. Grazie a questa nuova sintassi, non dobbiamo più usare quel brutto if alla fine.

```
#breakElse.py

numero = int(raw_input("Scrivi un numero > 1: "))

```

```
fattore = numero / 2
while fattore > 1:
    if numero % fattore == 0:
        print "Non e' primo: si puo' dividere per", fattore
        break
    fattore -= 1
else:
        print "E' primo!", fattore
```

4.3 FOR

Molto spesso un ciclo viene impiegato per scorrere gli elementi di una oggetto iterabile. Ad esempio:

```
#whileSeq.py

sequenza = ('uno', 'due', 'tre', 'quattro', 'cinque')

i = 0
while i<len(sequenza):
    print sequenza[i]
    i += 1
```

Questo sistema è un po' primitivo. Innanzitutto non sempre un oggetto iterabile è indicizzato con un numero (si pensi ad insiemi e mappe, ad esempio). E poi usare un contatore è veramente scomodo è brutto: bisogna scrivere continuamente sequenza[i], e ricordarsi dell'incremento, se si vogliono evitare cicli infiniti. Per semplificare le cose, Python prevede un bellissimo costrutto chiamato for.

4.3.1 Sintassi e uso

La sintassi di for è:

```
for indice in sequenza:
    istruzioni
```

Un ciclo for scorre tutti gli elementi di una sequenza e li assegna di volta in volta alla variabile (o tupla di variabili!) indice. Ecco lo stesso programma di prima, scritto con un ciclo for:

```
#for.py

for numero in ('uno', 'due', 'tre', 'quattro', 'cinque')
  print numero
```

Nota: anche per for è valido tutto il discorso già fatto per while sulle istruzioni pass, else, break e continue.

4.3.2 For e i dizionari

Dietro le quinte, il ciclo for usa gli iteratori, così come abbiamo visto nel paragrafo §3.8: questo gli permette di scorrere qualsiasi sequenza iterabile, come ad esempio un dizionario (che, come abbiamo visto, genera automaticamente un iteratore sulle chiavi):

```
capitali = {
  'Senegal': 'Dakar',
  'Ghana': 'Accra',
  'Togo': u'Lomè',
}
for stato in capitali:
  print u'la capitale del %s è %s' % (stato, capitali[stato])
```

Ma questo ciclo può essere ulteriormente migliorato, richiedendo al dizionario un itemiterator (vedi §3.8.2) e usando l'unpacking (vedi §3.1.6).

```
for (stato, capitale) in capitali.iteritems():
    print u'la capitale del %s è %s' % (stato, capitale)
```

4.3.3 For, range, xrange

Talvolta si vuole iterare su una sequenza di numeri. Conosciamo già un modo per farlo, in realtà: possiamo costruire una lista con la funzione range (vedi paragrafo §3.4.1). Ad esempio, per stampare i numeri da 1 a 10000, possiamo scrivere:

```
#nota: questo ciclo è perfettibile
for n in range(1,10001):
    print n
```

Questo sistema funziona, ma non è il massimo dell'ottimizzazione: prima di eseguire il ciclo for il programma deve costruire una lista di 10000 interi, e ciò richiede tempo e, soprattutto, molto spazio. In realtà non serve costruire una lista con ogni elemento: basterebbe generarne uno alla volta. Se avete seguito bene l'ultimo capitolo avrete già intuito la soluzione: un iteratore. Python mette a disposizione xrange, un oggetto capace di generare un iteratore su una sequenza di interi. Usandolo al posto di range evitiamo di sprecare inutilmente tempo e spazio:

```
for n in xrange(1,10001): #OK!
    print n
```

4.3.4 For e gli iteratori

For può essere usato con tutti gli iteratori, come quelli visti nel paragrafo §3.8. Alcuni di questi sono molto utili.
Immaginiamo, ad esempio, di dover scrivere avere una lista di cose da comprare al supermercato, come questa:

```
spesa = ['pane', 'acqua', 'supercomputer']
```

Vogliamo scrivere un ciclo che stampi gli articoli così:

```
0 – pane
1 – acqua
2 – supercomputer
```

Questo caso è più complicato di un 'semplice' ciclo for, perché dobbiamo tenere il conto dell'iterazione corrente. Potremmo usare una variabile, così:

```
i = 0
for i in len(spesa):

   print '%d - %s' % (i, spesa[i])
   i += 1
```

Ma in questo modo rispunta la variabile da incrementare, come nei cicli while. Possiamo fare di meglio, usando l'iteratore enumerate (vedi paragrafo §3.8.2) in congiunzione con l'unpacking (vedi paragrafo §3.1.6):

```
for (i, articolo) in enumerate(spesa):
   print '%d - %s' % (i, articolo)
```

Allo stesso modo, potete usare reversed per iterare su una sequenza dalla fine all'inizio, senza dover spendere tempo e spazio a crearne una rovesciata via slicing.

```
#bifronte.py
for lettera in reversed('enoteca'):
   print lettera,

C:\> python bifronte.py
a c e t o n e
```

4.4 SWITCH

Con for si conclude la carrellata dei costrutti per il controllo del flusso in Python. Chi viene da altri linguaggi può notare quanto il linguaggio sia parco nel fornire alternative (niente do…loop, goto, eccetera). Questo fa parte della filosofia di Python, riassunta nel motto: "dovrebbe esserci una – e preferibilmente una sola - maniera di fare le cose".

Fra i grandi assenti c'è anche il costrutto switch, talvolta malvisto anche nei linguaggi che lo supportano per la sua affinità al famigerato goto.

Un modo per simularlo è usare un dizionario di oggetti funzione. (Per comprendere bene l'esempio seguente è necessario aver letto il prossimo capitolo.)

```
#switch.py
#varie azioni (oggetti funzione)
def SePiovoso(): print "Prendo l'ombrello"
def SeSereno(): print "Mi vesto leggero"
def SeNevoso(): print "Metto le catene"
def Altrimenti(): print "Ok, grazie."

#associa le azioni al tempo
azioni = {
    'piovoso': SePiovoso,
    'sereno': SeSereno,
    'nevoso': SeNevoso
}

#chiede che tempo fa
tempo = raw_input(u"com'è il tempo, oggi?")

#esegue l'azione a seconda del tempo
azioni.get(tempo, Altrimenti)()
```

FUNZIONI E MODULI

Se siete arrivati fino a questo capitolo: complimenti! Ora siete già in grado di scrivere dei piccoli script di complessità limitata. Per riuscire a gestire progetti più importanti, però, dobbiamo fare un passo avanti e imparare a strutturare la nostra applicazione, in modo da poterla suddividere in parti logicamente distinte e riutilizzabili.
In questo capitolo vedremo come suddividere la programmazione in moduli e funzioni sia un ottimo modo per avvicinarci a quest'obiettivo.

5.1 FUNZIONI

Le funzioni non ci sono nuove: finora, infatti, abbiamo usato molte funzioni builtin, ovverosia quelle fondamentali di Python. Una funzione può prendere un certo numero di argomenti ingresso, e restituisce un valore. La funzione builtin len, per esempio, prende in ingresso una sequenza e ne restituisce la lunghezza.
In questo capitolo impareremo come definire funzioni tutte nostre.

5.1.1 Creare una funzione

La sintassi più semplice per definire una funzione è la seguente:

```
def NomeFunzione(argomento1, argomentoN, …):
    istruzioni
```

Per esempio, proviamo a creare dall'interprete una funzione saluta, che stampi a video 'ciao!'.

```
>>> def Saluta():
...    print 'ciao!'
...
```

Questa definizione viene letta da Python e interpretata, esattamente come qualunque altra istruzione (è un'istruzione come le altre). E' bene che impariate da questo istante a vederla come un assegnamento. Abbiamo creato un oggetto di tipo function e lo abbiamo assegnato alla variabile Saluta.

```
>>> Saluta
<function saluta at 0x01D55A70>
```

Ebbene sì: una funzione è un oggetto esattamente come gli interi, le tuple e le stringhe. Possiamo trattare Saluta proprio come faremmo con una qualsiasi altra variabile. Ad esempio, possiamo assegnare il suo oggetto ad un'altra:

```
>>> Ciao = Saluta
>>> del Saluta
>>> Ciao
<function Saluta at 0x01D55A70>
>>> Saluta
Traceback (most recent call last):

  File "<stdin>", line 1, in <module>
NameError: name 'Saluta' is not defined
```

Notate che il nome della variabile ora è Ciao, ma quello proprio dell'oggetto funzione continua ad essere Saluta. Se cancelliamo anche Ciao, la funzione verrà rimossa dalla memoria, esattamente come un oggetto qualsiasi.
A differenza di quelli analizzati finora, gli oggetti di tipo function

sono richiamabili (o callable): possono, cioè, essere eseguiti passando eventuali argomenti fra parentesi.

```
>>> def StampaQuadrato(x):
...     print x**2
...
>>> StampaQuadrato(2)
4
```

Ai programmatori C/C++/Java/Altro faccio notare ancora una volta che Python è tipizzato dinamicamente e implicitamente: si indica esclusivamente il nome degli argomenti, e non il tipo.

5.1.2 Return

Notate che la funzione che abbiamo scritto stampa il quadrato, non ne restituisce il valore. Tuttavia, ogni funzione in Python restituisce qualcosa al chiamante, perfino StampaQuadrato. Se non lo indichiamo esplicitamente, infatti, al termine della funzione Python restituisce automaticamente None: un'istanza del tipo NoneType, che rappresenta un valore 'nullo'.

```
>>> n = StampaQuadrato(2)
4

>>> print n
None
```

(Notate che durante l'assegnamento Python ha stampato 4. Per forza: abbiamo richiamato StampaQuadrato!)
Spesso le funzioni dovranno restituire un valore. Ad esempio, potremmo scrivere una funzione Quadrato che restituisca il quadrato di un numero. Per questo, dobbiamo usare l'istruzione return:

```
>>> def Quadrato(x):
...     return x**2
```

Abbiamo creato un punto di uscita dalla funzione: quando l'interprete incontra return, infatti, calcola l'espressione di ritorno (in questo caso x**2) e la restituisce al chiamante.

```
>>> n = Quadrato(2)
>>> print n
2
```

Niente vieta di scrivere più di un istruzione return:

```
#returnMultipli.py

def DescriviAltezza(metri):
    if metri >= 1.80:
        return 'alto'

    if metri <= 1.60:
        return 'basso'
    return 'normale'

altezza = float(raw_input('Quanto sei alto (in metri)?'))
print 'Sei', DescriviAltezza(altezza)
```

Notate la struttura della funzione: return è un punto di uscita, quindi se viene eseguito il secondo if significa che il primo è sicuramente fallito. E se si arriva alla fine della funzione si significa che 160 > altezza < 180, e possiamo restituire 'normale'. Se pensate che sia poco leggibile, potete modificarla in un if..elif..else (vedi paragrafo §4.1.2).
Potete restituire un solo valore, ma ricordatevi che state pro-

grammando in Python, e niente vi vieta di restituire una tupla, una lista o, ancor meglio, un dizionario contenente più valori!

5.2 SCOPE

I meccanismi che regolano in Python gli ambiti di visibilità (o scope) degli attributi hanno un'importanza cruciale: molti errori dei novizi derivano da una loro mancata comprensione. I programmatori esperti in altri linguaggi devono stare ancora più attenti, perché rischiano di fare analogie sbagliate: le regole di scope in Python funzionano molto diversamente che in C, C++ e Java.

5.2.1 Lo Scope Builtin

In Python gli scope sono fatti "a strati": dal più esterno al più interno. Lo scope più esterno è quello builtin. Quando scrivete:

```
print True
```

True non è definito da nessuna parte nel vostro programma, tuttavia Python lo trova: come fa? In realtà, quando si accorge che non avete definito niente che abbia il nome 'True', l'interprete estende la ricerca a i nomi builtin offerti da Python e presenti nel modulo __builtin__.
Niente vi vieta (per quanto sia una pratica discutibile) di creare una variabile con nome True.

```
>>> True = False
>>> True
False
```

Fate attenzione: in questo programma non abbiamo "cambiato valore alla variabile True, minando così alle fondamenta tut-

to l'impianto logico di Python". Abbiamo semplicemente creato una variabile nello scope corrente, che nasconde quella builtin. Per cambiare veramente il True builtin dovremmo assegnare a __builtin__.True.

```
>>> import __builtin__
>>> __builtin__.True = False
>>> True
False
```

Ovviamente il punto di questo paragrafo non è quello di istigarvi a corrompere i nomi builtin di Python (lasciateli perdere!). Il punto focale è che impariate che c'è differenza fra "fare riferimento a un nome" e "assegnare un nome".
Quando fate riferimento a un nome, mettete in moto tutto un meccanismo di ricerca (basato sulla regola LEGB, che vedremo) prima nello scope corrente, e poi in quelli esterni (via via, fino a quello builtin).
Quando assegnate un nome, ne state creando uno nuovo nello scope corrente, che nasconde qualsiasi nome presente in quelli più esterni.

5.2.2 Variabili globali e locali

Un assegnamento che avviene all'esterno di qualunque funzione crea una variabile globale. Le variabili globali si chiamano così perché è possibile far loro riferimento in qualsiasi punto del modulo.

```
#globale.py

pigreco = 3.14  #variabile globale

def Circonferenza(raggio):
```

```
    return 2 * raggio * pigreco
def AreaCerchio(raggio):
    return pigreco * raggio**2
```

Un assegnamento all'interno di una funzione, invece, crea sempre una variabile locale. La variabile locale è interna alla funzione: non viene vista, cioè, al di fuori della funzione in cui è stata creata.

```
#erroreDiScope.py

def CreaPiGreco():
    pigreco = 3.14 # variabile locale

CreaPiGreco()
print pigreco #errore!
```

Questo programma è sbagliato. Anche se CreaPiGreco viene richiamata, questo non crea una variabile pigreco globale. La variabile pigreco, infatti, è interna alla funzione: viene mantenuta finché questa è attiva e viene distrutta all'uscita.
L'istruzione print tenta di trovare un valore (pigreco) che non è visibile. Infatti questo è il risultato dell'esecuzione:

```
C:\> erroreDiScope.py
Traceback (most recent call last):
  File "erroreDiScope.py", line 6, in <module>

    print pigreco #errore!
NameError: name 'pigreco' is not defined
```

5.2.3 Global

Quando si fa riferimento ad un nome all'interno di una funzio-

ne, viene cercato prima fra le variabili locali, quindi fra le globali, quindi fra i builtin. Al solito l'assegnamento di una variabile all'interno di una funzione crea una nuova variabile che nasconde un'eventuale variabile globale. Questo codice lo dimostra:

```
#noGlobal.py
pigreco = 3.15

def CorreggiPiGreco():
    pigreco = 3.14

CorreggiPiGreco()
print pigreco

C:\> python noGlobal.py
3.15
```

L'esecuzione dimostra che il pigreco globale non è stato alterato: la variabile 'pigreco' interna a "CorreggiPiGreco" locale è un nome distinto che nasconde quello globale.
Questo è il comportamento standard di Python. Se, invece, si vuole davvero cambiare il valore di una variabile globale all'interno di una funzione, bisogna indicarlo esplicitamente all'interprete, dichiarando la variabile come global.

```
#global.py
pigreco = 3.15

def CorreggiPiGreco():
    global pigreco
    pigreco = 3.14

CorreggiPiGreco()
```

```
Print pigreco
```

In esecuzione, stavolta, pigreco verrà effettivamente cambiato, perché all'interno della funzione si assegnerà direttamente alla variabile globale.

```
C:\> python global.py
3.14
```

Fate attenzione al fatto che i nomi locali sono determinati staticamente, e non dinamicamente. Detto in altre parole: all'interno della funzione una variabile o è locale (e lo è per tutta la durata della funzione) o è globale (idem).
Un esempio spiegherà meglio:

```
#paradosso.py
pigreco = 3.15

def CorreggiPiGreco():

    print pigreco

    pigreco = 3.14

CorreggiPiGreco()
```

Questo programma è sbagliato. Quando Python legge la funzione (prima ancora di eseguirla) la compila in bytecode, vede l'assegnamento "pigreco = 3.14" e stabilisce che pigreco è una variabile locale.
Pertanto, quando in esecuzione si tenta di stampare pigreco ci si sta riferendo a una variabile locale, che però non è ancora "nata". Grande Giove!

In fase di esecuzione ciò si traduce in un'eccezione di tipo UnboundLocalError:

```
C:\> paradosso.py
Traceback (most recent call last):
  File "paradosso.py", line 8, in <module>
    CorreggiPiGreco()
  File "paradosso.py", line 5, in CorreggiPiGreco
    print pigreco
UnboundLocalError: local variable 'pigreco' referenced
                                        before assignment
```

5.2.4 Funzioni innestate

Abbiamo visto che le definizioni di funzioni sono molto simili a degli assegnamenti. E così, come potete assegnare una variabile all'interno di una funzione, potete anche definire una funzione all'interno di una funzione.

```
def FunzioneEsterna():
    def FunzioneInterna():
        print variabileEsterna

    variabileEsterna = 'Ciao'
    FunzioneInterna()

FunzioneEsterna()
```

Le funzioni interne hanno uno scope più interno, e stanno alla funzioni esterne proprio come le variabili locali alle globali, e le globali alle builtin. Quando Python deve risolvere un nome all'interno di una funzione cerca:

- Prima nel contesto locale

- Poi in tutte le funzioni esterne (se ne esistono)
- Poi nel contesto globale
- Quindi nei __builtins__

Questa regola di risoluzione viene spesso indicata con le iniziali di ogni contesto, nell'ordine: LEGB (Locali, Esterne, Globali, Builtin).
L'assegnamento, invece, genera sempre una nuova variabile nel contesto locale (a meno che non si dichiari esplicitamente la variabile come global, che crea/modifica una variabile globale).

5.3 ARGOMENTI

Il passaggio degli argomenti richiede qualche approfondimento: innanzitutto dobbiamo chiarire un equivoco tipico che porta a sbagliare molti neofiti.
Poi approfondiremo alcune sintassi alternative che permettono di passare argomenti predefiniti, e/o per nome, e/o di numero variabile.

5.3.1 Passaggio per assegnamento

Completiamo il discorso sugli scope parlando degli argomenti. Gli argomenti in Python vengono passati per assegnamento. In pratica viene creata una variabile locale che viene fatta puntare sull'oggetto passato. Cambiando il valore della variabile non modificate l'oggetto passato, ma la spostate semplicemente su un altro valore.
La piccola sessione qui sotto lo dimostra:

```
>>> def Incrementa(n):
...     n += 1
...
>>> dieci = 10
```

```
>>> Incrementa(dieci)

>>> dieci
10 #Sempre 10!
```

All'entrata in incrementa, viene assegnato: "n=dieci", così ora n e dieci puntano allo stesso oggetto (int(10)). Ma quando viene eseguito n += 1, il riferimento di n viene spostato su int(11). La variabile dieci, invece, rimane esattamente com'è.

5.3.2 Argomenti predefiniti

Python permette di scrivere una funzione del genere:

```
>>> def Saluta(messaggio='Ciao mondo!'):
...   print messaggio
```

La sintassi messaggio='Ciao a tutti' indica un argomento predefinito. Quando si richiama Saluta, le si può passare l'argomento messaggio, oppure no. In quest'ultimo caso, messaggio varrà automaticamente 'Ciao a tutti!'.

```
>>> Saluta("Felicissima sera…")
Felicissima sera…

>>> Saluta()
Ciao a tutti!
```

E' possibile mescolare parametri normali e predefiniti, purché questi ultimi vengano presentati alla fine (in caso contrario si creerebbero delle ambiguità facilmente immaginabili).

```
def Eleva(esponente=2, base) #NO!
def Eleva(base, esponente=2) #OK!
```

5.3.3 Associazione degli argomenti per nome

Immaginiamo di avere questa funzione per registrare delle persone in una rubrica telefonica:

```
def Registra(nome='N/A', cognome='N/A', telefono='N/A')

```

Python accetta le seguenti chiamate:

```

Registra() #nome='N/A', cognome='N/A', telefono='N/A'
Registra('Pinco') #cognome='N/A', telefono='N/A'
Registra('Pinco', 'Palla') #telefono='N/A'
Registra('Pinco', 'Palla', '01234')
```

Immaginiamo, però, di essere in una situazione in cui conosciamo solo il cognome e il numero di telefono di una persona, e vogliamo registrarli. Non possiamo, perché il cognome figura come secondo argomento, e dobbiamo per forza scrivere anche il primo. Possiamo cavarcela passando come primo argomento quello di default:

```
Registra('N/A', 'Palla', '01234')
```

Ma non è il massimo della chiarezza e della comodità. Python ci permette una via più semplice: possiamo passare un argomento indicandone il nome. La nostra chiamata diventa così:

```
Registra(cognome='Palla', telefono='01234')
```

La chiamata per nome ci permette, in altre parole, di alterare l'ordine degli argomenti che passiamo, indicando solo quelli che ci interessano. Associare gli argomenti per nome – anziché per posizione come avviene normalmente – rende il codice più

leggibile e semplifica casi come quello appena mostrato.

5.3.4 Definire funzioni con argomenti variabili

Python permette anche di scrivere funzioni che accettano un numero di argomenti variabile, che vengono inseriti automaticamente in una tupla. Esempio:

```
>>> def Scrivi(*parole):
...     for parola in parole:
...         print parola,
...
```

Qui l'asterisco prima dell'argomento parole indica che si tratta di una tupla, in cui verranno raccolti tutti gli argomenti passati alla funzione. Ora possiamo passare a Scrivi un numero di argomenti variabile:

```
>>> Scrivi('benvenuti', 'in Python', 2.5)
benvenuti in Python 2.5
```

Anche in questo caso si possono mescolare argomenti 'normali' con argomenti variabili, purché la tupla venga scritta alla fine, e indichi "tutti gli argomenti restanti":

```
>>> def Ripeti(volte, *parole):
...     for v in xrange(0, volte):
...         for parola in parole:
...             print parola,
...

>>> Ripeti(3, 'Hip-hip', 'hurrah!')
Hip-hip hurrah! Hip-hip hurrah! Hip-hip hurrah!
```

Analogamente, si possono raccogliere tutti gli argomenti passati per nome (vedi paragrafo §5.3.3) in un dizionario, usando un doppio asterisco:

```
>>> def Registra(nome, cognome, **dati):
...     print "Elenco dei dati di", nome, cognome
...     for chiave, dato in dati.iteritems():
...         print '%s: %s' % (chiave,  str(dato))
...

>>> Registra('Pinco', 'Palla', Telefono='01234', Via='Tal dei tali, 15')
Elenco dei dati di Pinco Palla
Via: Tal dei tali, 15
Telefono: 01234
```

5.3.5 Richiamare funzioni con la sintassi estesa

Nel richiamare una funzione Python permette una sintassi estesa, molto simile a quella vista nel paragrafo precedente per le definizioni: si possono impacchettare gli argomenti posizionali in una tupla preceduta da un asterisco, e quelli per nome in un dizionario preceduto da due asterischi. Ecco degli esempi:

```
>>> def Moltiplica(a, b): return a*b
…
>>> argomenti = (2,3)

>>> moltiplica(*argomenti) #uguale a Moltiplica(2,3)
6
```

Per fare un esempio più complesso, proviamo a richiamare la funzione Registra vista nel paragrafo precedente:

```
>>> persona = ('Pinco', 'Palla')
>>> dati = {'Telefono': '01234', 'Via': 'Tal dei tali, 15'}

>>> Registra(*persona, **dati)
Elenco dei dati di Pinco Palla

Via: Tal dei tali, 15
Telefono: 01234
```

5.4 PROGRAMMAZIONE FUNZIONALE

Python prende a prestito dal mondo dei linguaggi funzionali (un po' dal lisp, un po' da Haskell…) alcuni strumenti che possono risolvere molti problemi in maniera elegante e leggibile.

5.4.1 Lambda

A volte capita di dover creare funzioni tanto semplici e specifiche che non vale la pena di assegnar loro un nome né di spendere tempo definendole con def. In questi casi, si può creare una funzione lambda.
Con la parola chiave lambda si definisce una funzione, esattamente come con def. Solo, non si è obbligati a darle un nome e non è possibile estenderla su più righe: la funzione deve essere costituita soltanto da un'espressione. Ecco un esempio della sintassi:

```
>>> lambda a, b: a+b
<function <lambda> at 0x01D555B0>
```

Quella che abbiamo creato qui sopra è una funzione anonima che accetta due argomenti (a e b) e ne restituisce la somma (esattamente come operator.add – vedi il box al paragrafo §2.2).
Una funzione lambda può essere assegnata ad un nome, inse-

rita in un contenitore, o (molto più comunemente) passata ad una funzione.

5.4.2 Map

La funzione map(fn, seq) prende come argomenti una funzione fn (spesso definita come lambda) e una qualsiasi sequenza iterabile seq, e restituisce una lista in cui ogni per ogni elemento di seq viene applicata fn.

Ad esempio, se disponiamo di una lista di prezzi in euro:

```
>>> prezziInEuro = [120, 300, 200]
```

e vogliamo ottenerne una con i prezzi in lire, possiamo scrivere:

```
>>> prezziInLire = map(lambda x: x*1936.27, prezzi)
>>> prezziInLire
[193627.0, 580881.0, 1936270.0]
```

5.4.3 Zip

La funzione zip(seq1, seq2) prende come argomenti due sequenze iterabili e restituisce una lista in cui ogni elemento n è una coppia (seq1[n], seq2[n]).

Un impiego tipico di zip è la costruzione di dizionari. Se abbiamo:

```
>>> ordini = ['coniglio', 'topo', 'uomo']
>>> animali = ['lagomorfi', 'roditori', 'primati']
```

Il valore di zip(ordini, animali) è:

```
>>> zip(ordini, animali)
[('coniglio', 'lagomorfi'), ('topo', 'roditori'), ('uomo', 'primati')]
```

Questo è un valore perfetto da passare alla funzione dict per la

costruzione di un dizionario (vedi paragrafo §3.6.1).:

```
>>> ordine = dict(zip(ordini, animali))
>>> ordine
{'coniglio': 'lagomorfi', 'topo': 'roditori', 'uomo': 'primati'}
```

5.4.4 Reduce

La funzione reduce(fn, seq), “riduce” una qualsiasi sequenza iterabile seq, applicando la funzione fn cumulativamente, da sinistra a destra. La cosa è molto più facile da spiegare con un esempio che a parole. Ammettendo di avere una lista:

```
>>> numeri = [1, 2, 3, 4, 5, 6, 7, 8]
```

Possiamo ottenerne la sommatoria, o la produttoria usando reduce.

```
>>> reduce(operator.add, numeri)
15
>>> reduce(operator.mul, numeri)
120
```

Ciò che succede, in pratica, è che viene applicata la funzione ai primi due elementi (1 + 2), poi al risultato e al terzo ((1+2) + 3), poi al risultato e al quarto, (((1+2)+3)+4), e così via.

5.4.5 Filter

La funzione filter(fn, seq), restituisce una lista in cui figurano tutti gli elementi di seq, tranne quelli per cui bool(fn(elemento)) == False. Al solito, un esempio aiuterà a chiarire.

```
>>> numeri = [1, 2, 3, 4, 5, 6, 7, 8]
>>> numeriPari = filter(lambda x: x % 2 == 0, numeri)
```

```
>>> numeriDispari = filter(lambda x: x % 2 != 0, numeri)
>>> numeriPari, numeriDispari
([2, 4, 6, 8], [1, 3, 5, 7])
```

5.4.6 List comprehension

Python prende in prestito da Haskell uno strumento di grande espressività, che può sostituire le chiamate a map, a reduce, e ad altre funzioni viste in questo capitolo: le list comprehension.
Nella sua forma più semplice, una list comprehension è simile ad una funzione map: genera una lista applicando una funzione su una sequenza iterabile. La sintassi, però, è più semplice:

Ecco un esempio:

```
>>> prezziInEuro = [120, 300, 200]
>>> prezziInLire = [p*1936.27 for p in prezziInEuro]
```

Come vedete, una list comprehension (che va sempre circondata da parentesi quadre), è molto simile in apparenza ad un ciclo for. In effetti, la dinamica sottostante è la stessa.
Questo significa che è possibile utilizzare anche tutti i trucchi che abbiamo visto nel paragrafo §4.3, come l'unpacking per i dizionari.

```
>>> articoli = {'Scheda Video': 280, 'Monitor LCD': 120, 'Casse': 45}
>>> ['%s da %d euro' % (a, p) for a, p in articoli.iteritems()]
['Monitor LCD da 120 euro',
'Casse da 45 euro',
'Scheda Video da 280 euro']
```

Un'estensione della sintassi delle list comprehension permette di inserire ciascun elemento nella lista solo se passa un test. Si tratta di un'operazione di filtraggio del tutto simile a quella che

si otterrebbe usando filter.

```
>>> ['%s da %d euro' % (a, p) for a, p in articoli.iteritems() if p > 90]
['Monitor LCD da 120 euro', 'Scheda Video da 280 euro']
```

Qui l'aggiunta "if p > 90" vuol dire: "inserisci l'elemento soltanto se il prezzo è maggiore di 90 euro".
Un estensione ulteriore permette di collegare più list comprehension. Ad esempio, avendo queste squadre:

```
>>> squadre = ['Juventus', 'Sampdoria', 'Inter']
```

possiamo facilmente organizzare gli incontri con una list comprehension composta:

```
>>> ['%s contro %s' % (s1, s2) for s1 in squadre for s2 in squadre if
s1
!= s2]
['Juventus contro Sampdoria',
 'Juventus contro Inter',
 'Sampdoria contro Juventus',
 'Sampdoria contro Inter',
 'Inter contro Juventus',
 'Inter contro Sampdoria']
```

Notate anche l'uso di "if s1 != s2" per evitare che una squadra giochi contro sé stessa.
Provate, per esercizio, ad ottenere la stessa lista usando dei normali cicli for. Starà al vostro buon gusto decidere quale delle due scritture trovate più semplice o leggibile.

5.4.7 Generatori

Nel paragrafo §3.8 abbiamo visto gli iteratori. Finora abbiamo usato gli iteratori forniti da Python, come quelli generati dai

contenitori builtin, o da xrange. Tuttavia niente ci vieta di crearne di nostri. Il protocollo degli iteratori è davvero elementare: basta prevedere una funzione next e una __iter__, tuttavia c'è un metodo più semplice: creare un generatore.
Un generatore è una comunissima funzione che però non usa return per restituire un valore, bensì la parola chiave yield.

```
>>> def Contatore():
...     n = 0
...     while True:
...         n += 1
...         yield n
```

Possiamo richiamare la funzione Contatore per ottenere un generator object.

```
>>> i = Contatore()
```

Ora richiamiamo il metodo i.next(). Verrà eseguita la funzione che abbiamo scritto in def Contatore() e quando l'esecuzione arriverà a yield, next restituirà il valore n, proprio come un'istruzione return.

```
>>> i.next()
1
```

Ma a differenza di un return la funzione rimarrà in uno stato sospeso, proprio in quel punto. Un'altra chiamata ad i.next() risveglierà la funzione che continuerà, fino ad incontrare di nuovo lo yield, e così via.

```
>>> i.next(), i.next(), i.next(), i.next()
2, 3, 4, 5
```

Abbiamo creato un iteratore "infinito": usarlo in un'istruzione for senza una condizione di break non sarebbe un'idea tanto furba:

```
>>> for i in Contatore(): print i,
…
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 … CTRL + C
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
```

L'elenco andrà avanti all'infinito finché non ucciderete l'applicazione come abbiamo visto per il programma kamikaze (vedi paragrafo §4.2.1). Proviamo a modificare il ciclo in modo che conti da 0 a un numero stabilito dall'utente:

```
>>> def Contatore(max):
...     n = 0
...     while n < max:
...         n += 1
...         yield n
```

Questa nuova versione di Contatore prende un parametro che indica il massimo numero di iterazioni. Quando n >= max, il ciclo while verrà interrotto e la funzione finirà. A questo punto, verrà automaticamente generata un'eccezione di tipo StopIteration, che è proprio quello che ci si aspetta da un iteratore (vedi paragrafo §3.8).

```
>>> for i in Contatore(3): print i,
...
1, 2, 3
```

5.4.8 Generator Expression

Scrivere un generator è già un modo piuttosto semplice per creare un iteratore. Per funzionalità minime ce n'è uno ancora più semplice e affine alle list comprehension: le generator expression.
La sintassi di una generator expression è in tutto e per tutto simile a quella di una list comprehension, a parte il fatto che l'espressione va richiusa fra parentesi tonde, anziché quadre.
Anche la semantica è simile, a parte il fatto che viene prodotto un iteratore il cui metodo next() fornisce, uno alla volta, gli elementi dell'espressione.

```
>>> squadre = ['Juventus', 'Sampdoria', 'Inter']
>>> partite = ('%s contro %s' % (s1, s2) for s1 in squadre for s2
                                        in squadre if s1 != s2)
>>> partite.next()
'Juventus contro Sampdoria',
>>> partite.next()
 'Juventus contro Inter',
```

Le generator expression sono più indicate delle list comprehension in tutti quei casi in cui è richiesta un iterazione e non la creazione di un intera lista, come ad esempio nei cicli for. È il medesimo discorso che abbiamo affrontato nel capitolo su xrange (vedi paragrafo §4.3.3).

5.5 MODULI

Il meccanismo di divisione dei progetti in moduli è fondamentale in Python. Tanto fondamentale che l'abbiamo già sfruttato prima di affrontare questo capitolo. Abbiamo importato caratteristiche dal futuro, e funzioni dal modulo operator.
In questo capitolo vedremo sul serio cos'è un modulo, cosa significa importare e come creare dei moduli che possano esse-

re d'aiuto a noi stessi e agli altri.

5.5.1 Cos'è un modulo?

Finora abbiamo sempre operato all'interno di moduli, perfino quando abbiamo usato l'interprete interattivamente. Il nome del modulo che abbiamo usato finora (accessibile tramite l'attributo __name__) è '__main__'. Possiamo verificarlo facilmente:

```
>>> __name__
'__main__'
```

Lo scopo di un modulo è sostanzialmente quello di contenere dei nomi (o attributi). Potete vederlo come un grande spazio di nomi (o namespace) che vengono creati attraverso assegnamenti o definizioni di funzioni (globali), e finiscono in un attributo particolare chiamato __dict__.
Un __dict__ è un dizionario che rappresenta l'intero namespace del modulo sotto forma {nome: valore}. Ad esempio, se definiamo:

```
>>> pigreco = 3.14
```

Non avremo fatto altro che aggiungere al __dict__ del modulo '__main__' l'elemento: {'pigreco': 3.14}.

5.5.2 Importare un modulo

Un modulo deriva (normalmente) da un file .py. Quando si esegue un file dal prompt, ad esempio:

```
#pigreco.py
pi = 3.14
print 'Stiamo eseguendo pigreco.py!'

```

```
C:\> python pigreco.py
Stiamo eseguendo pigreco.py!
```

Viene cercato il file, letto, ed eseguito. Come abbiamo visto, un modulo eseguito direttamente prende il nome di '__main__'.
È anche possibile importare un modulo, grazie all'istruzione import.

```
>>> import pigreco
Stiamo eseguendo pigreco.py!
```

L'istruzione import pigreco ordina all'interprete: "trova il modulo pigreco.py, e se non l'hai mai importato prima, compilalo in un file .pyc ed eseguilo". Come si vede dalla risposta dell'interprete, Python esegue realmente il modulo, dopodiché il controllo ritorna al modulo chiamante ('__main__', in questo caso).
A questo punto, il nostro modulo ha acquisito un nuovo attributo: 'pigreco', che punta al modulo 'pigreco'.

```
>>> pigreco
<module 'pigreco' from 'pigreco.pyc'>
```

Ora possiamo accedere ad ogni nome all'interno di pigreco

```
>>> print pigreco.pi
3.14
```

Accedere ad un attributo di un modulo attraverso il punto vuol dire, in realtà, chiedere a Python il valore di un nome contenuto nel suo namespace __dict__. In altre parole, l'istruzione qui sopra è un modo più semplice per dire:

```
>>> print pigreco.__dict__['pi']
```

```
3.14
```

5.5.3 Elencare gli attributi di un modulo

Volendo, si può accedere all'intero contenuto del __dict__ di un modulo importato. È un buon modo per avere una panoramica di tutti i nomi definiti dal modulo, compresi quelli che vengono generati automaticamente da Python:

```
>>> for (nome, valore) in pigreco.__dict__.iteritems():
…   print nome, valore
…
__builtins__ {…}
__name__ pigreco
__file__ pigreco.py

pi 3.14
__doc__ None
```

Come vedete, è presente l'elemento {'pi': 3.14}, assieme ad altre voci che vengono aggiunte automaticamente al momento dell'import:

- __name__ indica il nome del modulo. Si può ottenere un riferimento ad un modulo a partire da un nome, consultando il dizionario sys.modules.
- __file__ indica il file del modulo (se esiste).
- __doc__ indica la docstring del modulo (lo vedremo al paragrafo §5.6) .
- __builtins__ non è altro che un riferimento al modulo __builtin__ (o al suo __dict__), contenente i nomi builtin offerti da Python.

Un modo più semplice per ottenere una lista dei soli nomi è usare la funzione dir(modulo):

```
>>> dir(pigreco)
['__builtins__', '__doc__', '__file__', '__name__', 'pi']
```

Quando dir viene usato senza parentesi, elenca gli elementi del modulo in cui ci si trova (ovverosia le chiavi di sys.modules[__name__].__dict__):

```
>>> dir()
['__builtins__', '__doc__', '__name__', 'pigreco']
```

5.5.4 Ricaricare un modulo

Come abbiamo visto, quando si importa un modulo questo viene cercato (all'interno del path, che è possibile modificare alterando la lista sys.path). Se non viene trovato, l'interprete solleva un'eccezione:

```
>>> import piRomano
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named piRomano
```

Se viene trovato il modulo, Python si preoccupa di vedere se il file è stato importato in precedenza. Se non è mai stato importato, Python compila il modulo in un file .pyc (ad esempio, pigreco.pyc), contenente le istruzioni in bytecode, e lo esegue.
Questo sistema permette di non sprecare tempo: se Python si accorge che il file è stato importato prima, non spende tempo a ricompilarlo e rieseguirlo.

```
>>> import pigreco #prima volta
Stiamo eseguendo pigreco.py
```

```
>>> import pigreco #seconda volta?
>>> #nulla!
```

Pertanto è bene abituarsi a non fare affidamento sul fatto che un modulo venga eseguito di seguito ad un import. Potrebbe capitare oppure no, a seconda di ciò che è successo prima. Se si vuole forzare una ricompilazione/riesecuzione (perché, per esempio, il modulo è stato modificato nel frattempo) si può usare la funzione reload(modulo).

```
>>> import pigreco #prima volta
Stiamo eseguendo pigreco.py
>>> reload(pigreco) #seconda volta!
Stiamo eseguendo pigreco.py
```

5.5.5 Sintassi estese di import

La sintassi di import permette qualche estensione rispetto a quanto mostrato finora: innanzitutto, si possono importare più moduli con una sola istruzione (la dinamica resta identica a quella già vista):

```
>>> import pigreco, operator, sys
```

Se il nome di un modulo è esageratamente lungo, o in conflitto con altri, si può fornirne un alias – ovverosia chiamarlo in un altro modo:

```
>>> import pigreco as pi

Stiamo eseguendo pigreco.py

>>> pi.pi
3.14
```

5.5.6 From

Il meccanismo di import ci permette di importare moduli che siamo obbligati a qualificare ogni volta. Ad esempio, siamo obbligati a scrivere:

```
>>> pigreco.pi
3.14
```

Per moduli e attributi dal nome molto lungo quest'obbligo può portare a gonfiare un po' il codice sorgente. Ad esempio:

```
>>> operator.add(pigreco.pi, math.sin(0))
3.1400000000000001
```

Python permette di snellire la scrittura importando soltanto alcune funzionalità di un modulo e copiandone il nome nel namespace di quello corrente. La sintassi è "from modulo import nome":

```
>>> from operator import add
>>> from pigreco import pi
>>> from math import sin
```

Ora il nostro esempio qui sopra si traduce semplicemente in:

```
>>>add(pi, sin(0))
```

C'è anche un ulteriore sintassi che permette di importare tutti i nomi di un modulo e ricopiarli di peso nel nostro: "from modulo import *"

```
>>> from math import *
>>> sin(10)**2 + cos(10)**2
1.0
```

5.5.7 Evitare collisioni

La sintassi con from sembra una grande idea solo all'inizio: l'entusiasmo passa quando si capisce che questa, in realtà, rischia di vanificare tutto il sistema dei namespace in Python, che permette di evitare pericolose situazioni di conflitto.
Nel nostro breve esempio abbiamo già creato un conflitto, infatti: il modulo math contiene un nome pi: scrivendo "from math import *" abbiamo sovrascritto il nostro pi acquisito prima con "from pigreco import pi".
In generale, cercate di non usare la sintassi con from e, soprattutto, import *: qualificare pienamente i nomi (pigreco.pi, math.pi), vi permette di evitare sgradite sorprese (è per questo che esistono i namespace!).
Quando scrivete un modulo, dovreste cercare di rendere visibili a from soltanto i nomi realmente utili al chiamante, e avete ben due sistemi complementari per farlo.
Metodo per esclusione: un nome globale che inizia per underscore ('_') non viene importato da import *. Ad esempio:

```
#nascostoCon_.py
pigreco = 3.14
_nascosto = 'Variabile invisibile a import *'

>>> from nascostoCon_ import *
>>> pigreco
3.14
>>> _nascosto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_nascosto' is not defined
```

Metodo per inclusione: potete inserire tutti i nomi visibili a import * in una lista di stringhe chiamata __all__. Se __all__ è

presente, tutti i nomi non inseriti al suo interno verranno nascosti a import *. Ad esempio:

```
#nascostoConAll.py
pigreco = 3.14
nascosto = 'Variabile invisibile a import *'
__all__ = ['pigreco']

>>> from nascostoConAll import *
>>> pigreco
3.14
>>> nascosto
Traceback (most recent call last):
NameError: name 'nascosto' is not defined
```

In ogni caso, ricordatevi che entrambi i metodi valgono solo per import *. È sempre possibile accedere ai membri nascosti attraverso un semplice import:

```
>>> import nascostoCon_, nascostoConAll
>>> nascostoCon_._nascosto, nascostoConAll.nascosto
('Variabile invisibile a import *', 'Variabile invisibile a import *')
```

In linea generale, Python non si basa mai sulla filosofia dell'impedire l'accesso a nomi ed oggetti: per questo non esiste un equivalente dei membri private di C++ e Java.

5.6 DOCSTRING

I commenti, che abbiamo introdotto nel primo capitolo, sono uno solo uno dei modi per documentare il codice. Python ne offre anche un altro: le docstring. Una docstring è una stringa, scritta all'inizio di un modulo, di una funzione o di una classe,

che ne descrive il comportamento.
Ecco un esempio:

```
'''Questo modulo esporta una sola funzione: mcd, per il calcolo

del massimo comun divisore. Viene solitamente usata
                              per semplificare i termini di una frazione.'''

def mcd(a, b):
    """mcd(a, b) -> int
```

Restituisce il massimo comun divisore fra gli interi a e b.

```
    #algoritmo di euclide
    while (b != 0):
        a, b  = b, a%b
    return a
```

Quando l'interprete legge una docstring in un oggetto, la fa referenziare automaticamente nell'attributo oggetto.__doc__. Alcuni tool fra cui pyDoc, che fa parte della libreria standard - sfruttano queste informazioni per generare documentazione in modo del tutto automatico e interattivo.
La funzione builtin help, ad esempio permette di ottenere un riassunto esauriente ed ordinato della struttura di un oggetto. Ecco come help presenta il nostro modulo mcd:

```
>>> import mcd
>>> help(mcd)
Help on module mcd:

NAME
    mcd
```

```
FILE
    mcd.py

DESCRIPTION
    Questo modulo esporta una sola funzione: mcd, per il calcolo
    del massimo comun divisore. Viene solitamente usata per
    semplificare i termini di una frazione.

FUNCTIONS
    mcd(a, b)
        mcd(a, b) -> int

        Restituisce il massimo comun divisore fra gli interi a e b.
```

5.7 ARGV

Finora per l'input ci siamo affidati a raw_input. Gli script, però, di solito funzionano in maniera più "silenziosa": gli argomenti vengono passati direttamente dalla riga di comando.
A questo proposito, il modulo sys fornisce l'attributo argv: una lista che contiene tutti gli argomenti passati da riga di comando.

```
#elencaArgomenti.py
import sys

print sys.argv

C:\>python elencaArgomenti.py 10 "Frase composta" ParolaSingola
['elencaArgomenti.py', '10', 'Frase composta', 'ParolaSingola']
```

Come potete vedere dall'esempio, argv[0] corrisponde solita-

mente al nome del file, mentre i restanti sono gli argomenti veri e propri.

5.8 IMPORTARE ED ESEGUIRE

Come abbiamo visto, possiamo usare un modulo in due modi: importandolo ed eseguendolo. Quando scriviamo un modulo è facile accorgerci in che caso siamo: basta verificare il valore di __name__:

```
#… Definizione attributi da esportare …

if __name__ == '__main__':
    # … istruzioniPerLoScript …
```

E' una pratica comune utilizzare questo idioma per realizzare moduli che possono essere usati per entrambi gli scopi. Ad esempio, possiamo trasformare il modulo mcd così (per brevità taglio tutte le docstring):

```
#mcd.py
def mcd(a, b):
    while (b != 0):
        a, b  = b, a%b
    return a

if __name__ == '__main__':
    from sys import argv
    if (len(argv) > 2):
        print mcd(int(argv[1]), int(argv[2]))
    else:
        print 'Numero di argomenti insufficiente'
```

In questo modo, si può usare mcd come uno script:

```
C:\> python mcd.py 15 6
30
```

Ma se mcd viene usato in un import, lo script non sarà eseguito:

```
>>> import mcd
>>> mcd.mcd(15, 6)
30
```

Questo sistema viene spesso usato per scrivere dei test che servano contemporaneamente a collaudare le funzionalità esportate e come esempio per chi dovesse leggere il file sorgente.

5.9 ANDARE AVANTI

In questo capitolo avete imparato le operazioni fondamentali che è possibile compiere con funzioni e moduli, anche se sono stati esclusi dalla trattazione gli argomenti più avanzati.
Per quanto riguarda le funzioni, ad esempio, su un buon testo di riferimento potrete facilmente trovare informazioni sui decorator, sull'uso di yield come espressione e sull'applicazione parziale delle funzioni, per esempio. Ma è difficile che vi servano se siete ancora dei principianti.Quando arriverete a scrivere applicazioni molto grandi, sicuramente dovrete anche studiare il sistema di package import in Python e l'alterazione diretta del path degli import (questi approfondimenti vi permetteranno di creare strutture gerarchiche di moduli, sfruttandone la divisione in sottodirectory). Essendo Python un linguaggio a tipizzazione dinamica è fondamentale un lavoro capillare di testing di ogni singolo modulo. Ci sono diversi strumenti per farlo, e il più

interessante è senz'altro il modulo standard doctest, che permette di verificare automaticamente se i componenti del modulo seguono il comportamento indicato nelle docstring.

CLASSI ED ECCEZIONI

Nel precedente capitolo abbiamo fatto un bel salto di livello: ora siamo in grado di creare non solo semplici script, ma anche moduli che contengono funzioni pronte per essere riutilizzate. L'ultimo passo in avanti che faremo in questo libro sarà imparare a creare dei tipi, proprio come str, tuple e list. In altre parole, vedremo i concetti di base per la definizione di classi e della programmazione orientata agli oggetti. Completeremo il quadro illustrando il meccanismo delle eccezioni in Python: cosa sono, come intercettarle e perché usarle.

6.1 CREARE UNA CLASSE

Creare una nuova classe è molto semplice: basta definirla usando la parola chiave class, così:

```
>>> class Persona(object): pass
```

La classe che abbiamo creato è un oggetto (ebbene sì: anche lei), che rappresenta la classe ‘Persona’:

```
>>> Persona
<class __main__.Persona at 0xb7d7d0ec>
```

Da molti punti di vista, una classe funziona come un modulo: ha un proprio namespace (vedi §5.5.1), che può essere letto e modificato a piacimento:

```
>>> Persona.ordine = ‘primati’
>>> Persona.ordine
‘primati’
```

6.2 ISTANZIARE UNA CLASSE

Ma la caratteristica unica delle classi è che queste possono essere richiamate come funzioni, per creare delle istanze:

```
>>> autore = Persona()
```

Ora 'autore' appartiene al tipo 'class Persona' (o anche: è una 'istanza' di Persona):

```
>>> autore
<__main__.Persona object at 0x0209F070>
>>> type(autore)
<class '__main__.Persona'>
```

Ogni istanza ha un suo namespace, che inizialmente e vuoto:

```
>>> autore.__dict__
{}
```

A che classe appartieni?

Un altro modo per risalire al tipo di un oggetto è accedere al suo attributo __class__:

```
>>> print autore.__class__
<class '__main__.Persona'>
```

Se vogliamo sapere se un oggetto appartiene ad una certa classe, possiamo richiamare la funzione 'isinstance':

```
#autore è un'istanza di Persona?
>>> isinstance(autore, Persona)
True
```

Ma può essere riempito e modificato a piacimento:

```
>>> autore.nome = 'Roberto'
>>> autore.cognome = 'Allegra'
```

E ora seguite attentamente la prossima istruzione

```
>>> autore.ordine
'primati'
```

Il nome 'ordine' non fa parte del namespace di 'autore'. Allora perché Python ha risposto 'primati'? Semplice: quando un nome non viene trovato nel namespace dell'istanza, viene sfogliato quello della classe. Se la cosa non vi sembra nuova, è un bene: si tratta di una regola di risoluzione dello scope analoga a quella vista in §5.2.2 per le variabili locali. Anche qui, un assegnamento in un istanza nasconde un'eventuale variabile della classe:

```
>>> autore.ordine = 'scrittori'
>>> autore.ordine
'scrittori'
>>> Persona.ordine
'primati'
```

6.3 METODI

Una classe può contenere dati o funzioni. Quest'ultime prendono il nome di 'metodi', e hanno delle caratteristiche tutte speciali.

```
>>> class Persona(object):
...     def Saluta(self):
```

```
...         print "Ciao, mi chiamo %s %s" % (self.nome, self.cognome)
```

Qui sopra abbiamo definito un metodo Saluta per la classe Persona, e una cosa salta subito all'occhio: self. Questo deve sempre essere scritto come primo argomento di un metodo, e farà riferimento all'istanza che ha generato la chiamata. Esempio:

```
>>> autore.nome = 'Roberto';
>>> autore.cognome = 'Allegra'

>>> autore.Saluta()
Ciao, mi chiamo Roberto Allegra
```

Come vedete, l'istanza acquisisce automaticamente il metodo Saluta (abbiamo visto il perché nel paragrafo precedente). Notate che abbiamo richiamato Saluta senza argomenti: 'self', infatti, viene passato da Python implicitamente, e in questo caso farà riferimento all'istanza 'autore'.
Se vogliamo rendere la cosa più esplicita, però, possiamo richiamare direttamente il metodo della classe:

```
>>> Persona.Saluta(autore)
Ciao, mi chiamo Roberto Allegra
```

E' esattamente la stessa cosa: si dice in gergo che autore.Saluta è un metodo 'bound' (cioè collegato ad una specifica istanza), mentre Persona.Saluta è un metodo 'unbound'. Entrambi sono oggetti: possono essere copiati, richiamati, eccetera, eccetera:

```
>>> salutaRoberto = autore.Saluta
>>> salutaRoberto()

Ciao, mi chiamo Roberto Allegra
```

6.4 __INIT__

Finora abbiamo creato un'istanza richiamando la classe come se fosse una funzione senza argomenti. Spesso può essere necessario (o utile) richiedere uno o più argomenti di inizializzazione. Ad esempio, creare una Persona passando il suo nome e cognome ci evita di dover scrivere due istruzioni di assegnamento. Per questo possiamo definire un metodo __init__:

```
>>> class Persona(object):
...     def __init__(self, nome, cognome):
...         self.nome = nome

...         self.cognome = cognome
...     # […] resto dell'implementazione […]
```

__init__ è un metodo speciale (un 'hook'): chi viene da altri linguaggi ama chiamarlo (impropriamente) 'costruttore'. In realtà __init__ viene richiamato quando l'istanza è già stata costruita, ed è quindi più un 'inizializzatore' del suo namespace. Ora non è più possibile creare una Persona senza argomenti:

```
>>> autore = Persona()
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>

TypeError: __init__() takes exactly 3 arguments (1 given)
```

Bisogna passare nome e cognome, che verranno immediatamente inseriti nel namespace dell'istanza:

```
>>> io = Persona('Roberto', 'Allegra')
>>> tu = Persona('Ignoto', 'Lettore')
>>> io.Saluta()
```

```
Ciao, mi chiamo Roberto Allegra
>>> tu.Saluta()
Ciao, mi chiamo Ignoto Lettore
```

6.5 EREDITARIETÀ

Una classe può essere scritta 'partendo da zero', oppure basandosi su un'altra. In quest'ultimo caso si dice che 'eredita' (o 'è una sottoclasse) da una 'classe base' (o 'superclasse'). La lista di tipi da cui ereditare viene indicata fra parentesi, di seguito al nome della classe. Non è una novità, in effetti: se controllate, vi accorgerete che finora abbiamo ereditato sempre dal tipo 'object', che aggiunge diverse caratteristiche interessanti alle nostre classi. (Le classi che non derivano da object vengono dette "vecchio stile", non permettono l'uso di funzionalità avanzate e si comportano un po' diversamente. Per questo consiglio sempre di ereditare da object oppure da una classe "nuovo stile", come Persona o un qualunque tipo builtin). Possiamo ereditare da qualsiasi classe: un Programmatore, ad esempio, a suo modo è una persona: ha un nome, un cognome e, quando non ha gravi problemi di socializzazione, saluta pure. Quindi possiamo derivare un Programmatore dalla classe Persona, così:

```
>>> class Programmatore(Persona): pass
```

Questo è sufficiente a far sì che un Programmatore si comporti come una Persona:

```
>>> io = Programmatore('Roberto', 'Allegra')
>>> io.Saluta()
Ciao, mi chiamo Roberto Allegra
```

In realtà quello che succede ogni volta che viene cercato un at-

tributo (scrivendo oggetto.attributo) è affine a quel che accade fra istanze e classi, e fra variabili locali e variabili globali. Le classi derivate hanno un loro namespace; se un attributo non viene trovato, Python lo cerca automaticamente nelle classi base, partendo da quella più prossima (Persona) per risalire passo passo tutta la gerarchia (fino a object). Il che vuol dire che un attributo della classe derivata nasconde (o in gergo 'sottopone ad override') eventuali attributi omonimi nella classe base. L'overriding è utile per 'personalizzare' il comportamento di una sottoclasse:

```
>>> class Programmatore(Persona):
...     def Saluta(self):
...         print 'Ciao, io sono %s %s, e programmo in Python' %
                                        (self.nome, self.cognome)
...
>>> bdfl = Programmatore(Guido', 'van Rossum')
>>> bdfl.Saluta()
Ciao, io sono Guido van Rossum, e programmo in Python
```

Chi sono i tuoi genitori?

L'elenco delle (eventuali) superclassi di un tipo è contenuto nella tupla Classe.__bases__.

```
>>> Programmatore.__bases__
(<class '__main__.Persona'>,)
```

La funzione issubclass(sottoclasse, superclasse) è un modo comodo per sapere se una classe deriva da un'altra:

```
#Programmatore deriva da Persona?
>>> issubclass(Programmatore, Persona)
```

```
True
>>> issubclass(io.__class__, Persona)
True
```

Molto spesso è meglio usare issubclass rispetto ad isinstance, dato che solitamente un comportamento che è accettabile per una classe lo è anche per le sottoclassi (vedi, ad esempio §6.6.2)

Non esistono al mondo solo programmatori Python, quindi sarebbe bello inizializzare un Programmatore anche con un terzo argomento: linguaggio. È facile: basta scrivere un __init__ con tre argomenti. Nel corpo del metodo, però, ci conviene richiamare l'__init__ (unbound) di Persona, così:

```
>>> class Programmatore(Persona):

...     def __init__(self, nome, cognome, linguaggio):

...         Persona.__init__(self, nome, cognome)
...         self.linguaggio = linguaggio
...     def Saluta(self):

...         print 'Ciao, io sono %s %s, e programmo in %s' %
                              (self.nome, self.cognome, self.linguaggio)
…
>>> ilVenerabile = Programmatore('Bjarne', 'Stroustrup', 'C++')
>>> ilVenerabile.Saluta()

Ciao, io sono Bjarne Stroustrup, e programmo in C++
```

Vi ritroverete spesso ad usare questo meccanismo di 'delega alla classe base' nei vostri metodi __init__ (e non solo in quelli!).

6.6 METODI SPECIALI

Come abbiamo visto nel capitolo 2, Python prevede molti tipi numerici builtin, ma non un tipo "Fraction", che rappresenti una frazione. Perché non realizzarlo noi? Iniziare è facile e divertente:

```
>>> class Frazione(object):
...     def __init__(self, num, den=1):
...         self.num, self.den = num, den
...         self.Semplifica()

...     def Semplifica(self):
...         import mcd #vedi §5.8
...         divisore = mcd.mcd(self.num, self.den)
...         self.num //= divisore
...         self.den //= divisore
```

Fin qui, nulla di nuovo. Ora possiamo creare delle frazioni:

```
>>> numero = Frazione(5,20)
>>> numero
<__main__.Frazione object at 0x020A7FD0>
>>> numero.num, numero.den
(1, 4)
```

6.6.1 Conversioni

Prima o poi l'utente della nostra classe avrà bisogno di sapere 'quanto vale' la sua Frazione espressa in un numero decimale. Il modo più naturale di offrire quest'operazione è definire una conversione verso il tipo float. In Python si può permettere una conversione verso ogni tipo numerico, semplicemente definendo il rispettivo metodo __tiponumerico__. Nel nostro caso: __float__.

```
#vedi §2.2.2
from __future__ import division
class Frazione(object):
# [… resto dell'implementazione …]
    def __float__(self):
        return self.num / self.den
```

Ora possiamo esprimere facilmente una Frazione con un numero decimale:

```
>>> print float(Frazione(10,7))
1.42857142857
```

La rappresentazione 'normale', però, ora suona molto scomoda: sarebbe bello scrivere 'Frazione(2/8)' nell'interprete e sentirsi rispondere '1/4', invece di <__main__.Frazione eccetera…>. Per migliorare la situazione Python definisce ben due metodi: __repr__ e __str__:

```
>>> class Frazione(object):
... # […resto dell'implementazione …]
...    def __repr__(self):

...        return '%s / %s (%f)' % (self.num, self.den, float(self))
...     def __str__(self):
...         return '%s/%s' % (self.num, self.den)
```

__repr__ viene invocato quando viene richiesta una 'rappresentazione' dell'oggetto (cioè quando si richiama la funzione 'repr', oppure, più semplicemente, quando si richiede il valore direttamente all'interprete). Bisognerebbe comunicare informazioni utili per parser e programmatori e poco "amichevoli per l'utente":

```
>>> numero = Frazione(20,5)
>>> numero
1 / 5 (0.200000)
```

__str__, invece, viene invocato quando viene richiesta una conversione a stringa dell'oggetto (cioè quando viene invocata la funzione 'str', oppure, più semplicemente quando si usa l'istruzione print). Bisognerebbe comunicare solo informazioni "amichevoli per l'utente":

```
>>> str(numero)
1/5
>>> print numero
1/5
```

6.6.2 Altri operatori

__init__, __str__ e __repr__ sono solo alcuni casi particolari di overloading degli operatori in Python. Metodi simili vengono detti 'hook' e sono richiamati automaticamente quando vengono effettuate determinate operazioni (come l'inizializzazione, la conversione a stringa e la rappresentazione di un oggetto). Molti degli operatori elencati nelle tabelle §2.1, §2.2 e §2.3, ad esempio, possono essere intercettati da uno hook nella forma __operatore__ (dove 'operatore' è il nome dell'operatore così come riportato dalla prima colonna di ogni tabella). Possiamo approfittarne, ad esempio, per permettere la moltiplicazione fra due frazioni, usando l'hook __mul__:

```
# [...] resto dell'implementazione [...]
def __mul__(self, other):
...   return Frazione(self.num*other.num, self.den*other.den)
>>> print Frazione(2*10) * Frazione(10/2)
1/1
```

Notate che la moltiplicazione funziona fra due Frazioni. Volendo, possiamo ritoccare il metodo __mul__ per fargli moltiplicare una Frazione con un numero:

```
def __mul__(self, other):
        if not issubclass(other.__class__, Frazione):
            other = Frazione(other)
        return Frazione(self.num*other.num, self.den*other.den)
```

Ora, in a*b, se b non è una Frazione (o un suo derivato. Per questo è meglio usare issubclass rispetto a isinstance), il metodo proverà a considerarlo come una frazione "b/1" (se non è neanche un numero, la moltiplicazione genererà un'eccezione). Ora possiamo scrivere senza problemi istruzioni del genere:

```
>>> n = Frazione(2,5) * 2
>>> print n
4/5

>>> n *= 2 #automaticamente generato da Python
>>> print n
8/5
```

Però non è possibile invertire i fattori: quest'istruzione genera un'eccezione:

```
>>> print 2 * Frazione(2,5)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'int' and 'Frazione'
```

Il perché è piuttosto semplice: Python richiama il metodo __mul__ dell'operando di sinistra – nel nostro caso int.__mul__, che

non sa come comportarsi con le Frazioni (e infatti lancia un eccezione di tipo TypeError).
Per rimediare, possiamo definire il metodo Frazione.__rmul__.
Quella "r" all'inizio del nome indica il fatto che il metodo verrà richiamato se l'operando 'destro' è di tipo Frazione. Tutto quello che dobbiamo fare all'interno di __rmul__, quindi, è richiamare __mul__!

```
def __rmul__(self, other):
    return self * other
```

Ora possiamo tranquillamente moltiplicare un intero per una Frazione:

```
>>> print 2 * Frazione(2,5)
4/5
```

Questo sistema vale per tutti gli operatori aritmetici binari: nel caso in cui l'operando di sinistra non preveda un metodo "__operatore__", Python richiamerà l' "__roperatore__" su quello di destra.

6.7 ATTRIBUTI 'NASCOSTI'

Ora la classe Frazione comincia a prendere forma. Uno degli aspetti più interessanti è che un oggetto di tipo Frazione viene semplificato automaticamente.

```
>>> n = Frazione(10, 5)
>>> print n
2/1
>>> n *= Frazione(1, 2)
1/1
```

L'utente della classe, quindi non ha bisogno di richiamare Semplifica() direttamente. Non ha neanche bisogno di sapere che esiste. In linguaggi come Java o C++ questo metodo sarebbe reso 'privato': completamente invisibile e inaccessibile. Questo concetto in Python non esiste: tutto è visibile, se si vuole. Ma ci sono modi per 'nascondere' un po' gli attributi, in modo che non si faccia confusione. Quando un attributo inizia (ma non finisce) con due underscore (__attributo), Python fa sì che all'esterno della classe questo sia visto come _Classe_Attributo. Se rinominiamo 'Semplifica' così:

```
def __Semplifica([… eccetera …]
```

Non sarà più accessibile:

```
>>> Frazione.Semplifica

AttributeError: type object 'Frazione' has no attribute 'Semplifica'
>>> Frazione.__Semplifica

AttributeError: type object 'Frazione' has no attribute '__Semplifica'
```

L'unico modo di accedervi sarà scrivendo:

```
>>> Frazione._Frazione__Semplifica

<unbound method Frazione.__Semplifica>
```

Questo sistema, dunque, non rende affatto "privato" un attributo, ma lo nasconde un po' agli utenti della vostra classe e soprattutto lo rende 'unico' agli occhi di Python (risolvendo così eventuali ambiguità che possono presentarsi in gerarchie complicate).

6.8 PROPRIETÀ

La semplificazione automatica non funziona se all'utente è concesso di ritoccare direttamente il numeratore o il denominatore della classe:

```
>>> n = Frazione(10, 5)
>>> n.den = 2
>>> print n
2/2 #eh?
```

Per controllare l'accesso a questi attributi dobbiamo trasformarli in 'proprietà'. Le proprietà sono attributi con una caratteristica speciale: quando vengono lette, modificate o rimosse, richiamano automaticamente la funzione che gli viene passata al momento della costruzione. Un esempio aiuterà a chiarire:

```
class Frazione(object):
    def __GetNum(self):
        return self.__num
    def __GetDen(self):
        return self.__den
    def __init__(num, den=1)
        self.__num, self.__den = num, den
        self.__Semplifica()
    den = property(GetDen)
    num = property(GetNum)

 [… resto dell'implementazione …]
```

Qui abbiamo scritto due metodi 'getter' (__GetNum e __GetDen), per restituire rispettivamente il valore del numeratore e del denominatore (notate che sia i metodi che gli attributi ora sono "nascosti", perché servono solo alla nostra classe). 'num' e 'den'

diventano ora due ‘proprietà’ (ossia: due oggetti di tipo property), a cui associamo i metodi getter (volendo potremmo associarne anche uno per l’impostazione e uno per la rimozione, scrivendo qualcosa del tipo: property(GetNum, SetNum, DelNum)). Apparentemente il comportamento della classe è invariato:

```
>>> n = Frazione(10, 5)
>>> print n.num, n.den
2 1
```

Ma ora, dato che non abbiamo impostato alcun metodo per la modifica, non è più possibile assegnare direttamente al numeratore e al denominatore (a meno di non andarsela a cercare ritoccando n._Frazione__num e n._Frazione__den):

```
>>> n.den = 2
Traceback (most recent call last):
AttributeError: can't set attribute
```

Grazie alle proprietà possiamo quindi controllare l’accesso/modifica/rimozione degli attributi senza per questo complicarci la vita usando direttamente metodi getter e setter.

6.9 ECCEZIONI

Ora possiamo copiare l’implementazione della nostra Frazione in un file (Frazione.py) in modo da poterla esportare come modulo. Frazione, in effetti, è diventata una bella classe: gli oggetti si comportano e vengono rappresentati in maniera semplice e naturale. Finche tutto va bene. Peccato che la vita sia fatta anche di casi particolari.
Proviamo ad aggiungere al nostro file qualche test (vedi 5.8):

```
#Frazione.py
class Frazione(object):
    # [… implementazione …]
if __name__ == '__main__':
    f = Frazione(10/0)
    print repr(f)
    print 'Test riusciti!'
```

Se eseguiamo lo script, vediamo che il test fallisce: il programma termina prematuramente, e Python lascia questo lungo messaggio:

```
Traceback (most recent call last):
  File "Frazione.py", line 48, in <module>
    print repr(f)
  File "Frazione.py", line 44, in __repr__
    return '%s / %s (%f)' % (self.num, self.den, float(self))
  File "Frazione.py", line 41, in __float__
    return self.__num / self.__den
ZeroDivisionError: float division
```

6.9.1 Propagazione delle eccezioni

Nel corso di questo libro ho presentato spesso messaggi del genere, e ho liquidato la cosa definendola "un'eccezione". Qui, finalmente, vedremo che si tratta in realtà del risultato finale di un lungo processo di 'propagazione delle eccezioni'. Python mantiene una pila (o 'stack') per gestire le chiamate a funzione, e il messaggio fornitoci dall'interprete (detto 'traceback') l'ha descritta precisamente. Leggendo, capiamo cos'è successo: abbiamo richiamato la funzione repr(Frazione(10, 0)), e questa contiene una riga in cui si richiede esplicitamente una conversione in float, e il metodo __float__ tenta dividere 10 per 0. Quest'operazione per Python è insensata, e provoca il sollevamento di

un'eccezione di tipo ZeroDivisionError. Al momento del sollevamento dell'eccezione, Python cerca un 'gestore' all'interno della funzione corrente (Frazione.__float__). Chiaramente non c'è, perché non sappiamo ancora cosa sia un gestore. Allora l'interprete 'srotola' lo stack e prova a vedere se c'è un gestore in __repr__. Non c'è. E via srotolando, fino a tornare a livello del modulo. Poiché nessuno dei chiamanti ha gestito l'eccezione, ci pensa Python, col comportamento predefinito: stampa il traceback e termina inesorabilmente l'applicazione. Questo sistema è molto utile e robusto: il traceback ci permette di seguire l'intera storia che ha portato al fattaccio, e l'uscita dal programma fa sì che non si propaghi una situazione scorretta, che potrebbe portare a comportamenti scorretti dell'applicazione e a bug difficili da identificare.
La parola chiave è: prevenire.

6.9.2 Lanciare un'eccezione

Quindi Frazione si comporta bene. Ma se bisogna davvero 'prevenire', si può anche far meglio: l'eccezione, infatti, viene lanciata solo quando si tenta una conversione a float (direttamente o indirettamente), e questo è spiazzante: l'utente giustamente si chiede: "perché finora ha funzionato, mentre se chiedo all'interprete il valore del mio oggetto (operazione normalissima) viene generata un'eccezione?". Il problema è a monte: non dovrebbe essere permesso creare una Frazione con denominatore pari a 0.
Sta a noi, quindi, lanciare un'eccezione direttamente in Frazione.__init__, usando l'istruzione 'raise'. Dobbiamo indicare il tipo dell'eccezione e, se vogliamo, un dato extra che chiarisca meglio cos'è successo.

```
class Frazione(object):
#[… resto dell'implementazione …]
```

```
def __init__(self, num, den=1):
        if den == 0:
            raise ZeroDivisionError, 'Frazione con Denominatore nullo'
        self.__num, self.__den = num, den
        self.__Semplifica()
```

Ora il nostro programma si accorge dell'errore immediatamente, e lascia un traceback molto più corto e significativo:

```
Traceback (most recent call last):
  File "Frazione.py", line 47, in <module>
    f = Frazione(10, 0)
  File "Frazione.py", line 32, in __init__
    raise ZeroDivisionError, "Frazione con denominatore nullo"
ZeroDivisionError: Frazione con denominatore nullo
```

Come tipo di eccezione abbiamo usato ZeroDivisionError, ma se vi sembra inappropriato, potete creare una vostra classe, eventualmente derivando dalle eccezioni già esistenti (ad esempio, ArithmeticError, da cui derivano tutti gli errori aritmetici, ZeroDivisionError compreso).

```
class ZeroDenominatorError(ArithmeticError): pass
# [… in Frazione.__init__ …]
if (den == 0): raise ZeroDenominatorError
```

In questo modo l'errore diventa tanto autoesplicativo da rendere superflua la descrizione extra.

6.9.3 Gestire un'eccezione

Proviamo ad usare la nostra classe Frazione in un'applicazione:

```
#gestori.py
```

```
import Frazione
while(True):

    num = int(raw_input("Numeratore:"))
    den = int(raw_input("Denominatore:"))
    f = Frazione.Frazione(num, den)
    print "%s valgono (circa) %f)" % (f, f)
```

Questa funziona bene in condizioni normali, ma non in quelle eccezionali. Sappiamo che se l'utente inserisce un denominatore nullo, ad esempio, verrà lanciato un Frazione.ZeroDenominatorError. Lasciare che questa uccida l'applicazione non è utile: è meglio intercettare l'eccezione e comunicare all'utente quanto è scemo. Per questo usiamo il costrutto try…except:

```
while(True):
    try:
        num = int(raw_input("Numeratore:"))
        # [… eccetera …]
    except Frazione.ZeroDenominatorError:
        print "Il denominatore dev'essere maggiore di zero"
```

Ora l'interprete proverà ad eseguire le istruzioni dentro il blocco try: se queste generano un'eccezione di tipo ZeroDenominatorError, questa verrà intercettata dal gestore (il meccanismo di ricerca è descritto in §6.9.1: i gestori che vengono incontrati prima nello srotolando lo stack 'rubano' l'eccezione a quelli che arrivano dopo). L'applicazione non muore più, ma esegue il codice del gestore e continua felice la sua esecuzione saltando il resto del blocco try…except. Possiamo scrivere più gestori, che verranno provati uno dopo l'altro. Nel nostro esempio, possiamo prevedere anche il caso in cui l'utente scriva una stringa (Numeratore: 'tre') al posto di un numero, generando così un'ec-

cezione di tipo 'ValueError'.

```
try:
    # [… istruzioni …]
except Frazione.ZeroDenominatorError:
    print "Il denominatore dev'essere maggiore di zero"
except ValueError:
    print "Ma allora sei *veramente* scemo!"
```

Un gestore viene considerato compatibile quando l'eccezione è del tipo indicato, oppure una sua sottoclasse. Questo permette di sfruttare le gerarchie per gestire 'tutti gli errori di un certo genere'. Ad esempio:

```
try:
    # [… istruzioni …]
except Frazione.ZeroDenominatorError:
    print "Il denominatore dev'essere maggiore di zero"
except ValueError:
    print "Ma allora sei *veramente* scemo!"
except ArithmeticError:
    print "Non riesco a calcolarlo"
```

Se si verificherà un errore aritmetico di qualsiasi tipo (come un overflow, ad esempio), l'applicazione dirà "non riesco a calcolarlo". Notate che, se il numeratore sarà nullo verrà richiamato solo il gestore di ZeroDenominatorError, nonostante questo derivi da ArithmeticError: i gestori vengono provati in sequenza e il primo compatibile 'ruba' l'eccezione ai successivi (per questo vanno messi prima i gestori delle sottoclassi, e poi quelli 'generici' delle superclassi). Il massimo della genericità è il gestore universale: un except da solo:

```
try:
```

```
    # [… istruzioni …]
#[…altri gestori …]
except:
    print "Niente da dire: sei scemo. Riprova."
```

In questo caso, qualunque eccezione verrà catturata da except, pertanto questo gestore va messo per ultimo, o gli altri diventeranno completamente inutili. Come tutte le cose che sembrano un'idea comoda e furba agli inizi, il gestore universale va evitato il più possibile. Si rischia davvero di vanificare tutto il sistema delle eccezioni, lasciando passare in modo silenzioso errori di tipo potenzialmente letale: in Python perfino gli errori di sintassi sono eccezioni!

6.9.4 Eccezioni e protocolli

Potrei usare questo paragrafo per parlarvi delle estensioni dei gestori: ovverosia di finally, else, e della futura parola chiave with. Non lo farò: sono argomenti importanti ma avanzati, per i quali vi rimando a un manuale di riferimento (possibilmente aggiornato). Preferisco usare questo paragrafo per una questione più fondamentale: chiarire il rapporto fra operazioni sui tipi ed eccezioni. Poiché la tipizzazione è implicita e dinamica, in Python non saprete mai in anticipo il tipo degli oggetti con cui avrete a che fare. Potrete certamente controllarlo (oggetto.__class__), ma non vi serve realmente saperlo: basta che certe operazioni siano definite. In altre parole: in Python la programmazione è orientata ai protocolli, non alle interfacce (se venite dal C++, immaginate di avere a che fare con dei template – molto più amichevoli e flessibili e, soprattutto, dinamici). Facciamo un esempio. Abbiamo questa funzione:

```
def Presenta(tizio, caio):
    tizio.Saluta()
```

```
    caio.Saluta()
```

Ora: a che tipo devono appartenere tizio e caio? Senz'altro, se limitiamo la funzione alle sole Persone (vedi §6.3) non ci saranno problemi.

```
if isinstance(tizio, Persona) …
else:
    print "Sarebbe imbarazzante!"
```

Ma così facendo tagliamo inesorabilmente fuori i Programmatori, che pure derivano da Persona (vedi §6.5) e quindi saprebbero salutare. Possiamo usare issubclass, invece di isinstance. Ma così tagliamo fuori i Robot. Un Robot, infatti, non è una Persona: è una classe definita così:

```
class Robot(object):
    def Saluta(self):
        print "uno è lieto di poter servire"
```

Robot, quindi non 'deriva' da Persona ma implementa il 'protocollo di saluto'. Per non discriminare nessuno basta gestire l'eccezione AttributeError, che si verifica quando si tenta di accedere ad un attributo di cui un oggetto non dispone:

```
def Presenta(tizio, caio):
    try:
        tizio.Saluta()
        caio.Saluta()
    except AttributeError:
        print "Sarebbe imbarazzante!"
```

Ora Presenta farà conoscere tranquillamente un Robot ad un

Programmatore, ma non introdurrà un Frigorifero a un Tostapane. (Se vi spaventa che la prima operazione possa andare a buon fine e la seconda no, potete accedere prima ai metodi memorizzandoli come bound (t,c = tizio.Saluta, caio.Saluta) e richiamarli successivamente (t(); c()).

6.10 ANDARE AVANTI

Qui abbiamo analizzato solo alcuni degli aspetti fondamentali di classi ed eccezioni, ma Python ha molto altro da offrire.
Solo per fare qualche esempio: una classe può derivare da più tipi (ereditarietà multipla), e questo permette tutta una serie di idiomi, problemi e soluzioni. Si possono implementare dei descrittori, che alterino l'accesso agli attributi (le proprietà sono un esempio di descrittori, infatti). Si possono creare classi a tempo di esecuzione e addirittura classi di classi (metaclassi) che modifichino il comportamento standard dei tipi. Eccetera, eccetera. Si tratta, comunque, di funzionalità avanzate. Se siete nuovi alla programmazione OOP è senza dubbio più importante che leggiate qualche classico (Design Patterns è storicamente il più indicato) che vi introduca alla progettazione a oggetti. E, ovviamente, niente può sostituirsi alle lezioni insegnate dalla pratica.